Name

intro – introduction to library functions

Description

This section describes functions that may be found in various libraries. The library functions are those other than the functions that directly invoke ULTRIX system primitives, described in section 2. Section 3 has the libraries physically grouped together. The functions described in this section are grouped into various libraries:

Sections 3 and 3s

The (3) functions are the standard C library functions. The C library also includes all the functions described in Section 2. These routines are included for compatibility with other systems. In particular, a number of system call interfaces provided in 4.2BSD have been included for source code compatibility. The (3s) functions comprise the standard I/O library. Together with the (3n), (3xti), (3yp) and (3) routines, these functions constitute library *libc*, which is automatically loaded by the C compiler (cc), the Pascal compiler (pc), and the FORTRAN compiler (f77). (FORTRAN and Pascal are optional and may not be installed on your system.) Declarations for these functions may be obtained from the include file, <stdio.h>. The link editor `ld`(1) searches this library under the –lc option. Declarations for some of these functions may be obtained from include files indicated on the appropriate pages.

VAX Only

On VAX machines, the GFLOAT version of *libc* is used when you use the `cc`(1) command with the **–Mg** option, or you use the `ld`(1) command with the **–lcg** option. The GFLOAT version of *libc* must be used with modules compiled with `cc`(1) using the **–Mg** option.

Note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and your program may produce erroneous results if this occurs on a VAX machine.

Section 3cur

The (3cur) library routines make up the X/Open curses library. These routines are different from the 4.2BSD curses routines contained in Section 3x.

Section 3f

The (3f) functions are all functions callable from FORTRAN. These functions perform the same jobs as do the (3) functions. An unsupported FORTRAN compiler, f77, is included in the VAX distribution. FORTRAN is available as a layered product on both VAX and RISC machines.

Section 3int

The (3int) functions assist programs in supporting native language interfaces. They are found in the internationalization library *libi*.

Section 3krb

The library of routines for the Kerberos authentication service. These routines support the authentication of commonly networked applications across machine boundaries in a distributed network.

Section 3m

The (3m) functions constitute the math library, *libm*. They are automatically loaded as needed by the Pascal compiler (pc) and the FORTRAN compiler (f77). The link editor searches this library under the **–lm** option. Declarations for these functions may be obtained from the include file, < math.h >.

VAX Only

On VAX machines, the GFLOAT version of *libm* is used when you use the `ld`(1) command with the **–lcg** option. Note that you must use the GFLOAT version of *libm* with modules compiled using the `cc`(1) command with the **–Mg** option.

Note that neither the compiler nor the linker `ld`(1) can detect when mixed double floating point types are used, and the program may produce erroneous results if this occurs on a VAX machine.

Section 3ncs

This section describes the NCS (Network Computing System) library routines. The Title, Name, and See Also sections of the NCS reference pages do not contain the dollar ($) sign in the command names and library routines. The actual NCS commands and library routines do contain the dollar ($) sign.

Section 3n

These functions constitute the internet network library,

Section 3x

Various specialized libraries have not been given distinctive captions. Files in which such libraries are found are named on appropriate pages.

Section 3xti

The X/Open Transport Interface defines a transport service interface that is independent of any specific transport provider. The interface is provided by way of a set of library functions for the C programming language.

Section 3yp

These functions are specific to the Yellow Pages (YP) service.

Environmental Compatibility

The libraries in Sections 3, 3m, and 3s contain System V and POSIX compatibility features that are available to general ULTRIX programs. This compatibility sometimes conflicts with features already present in ULTRIX. That is, the function performed may be slightly different in the System V or POSIX environment. These features are provided for applications that are being ported from System V or written

for a POSIX environment.

The descriptions in these sections include an ENVIRONMENT section to describe any differences in function between System V or POSIX and the standard C runtime library.

The System V compatibility features are not contained in the standard C runtime library. To get System V-specific behavior, you must specify that the System V environment is to be used in compiling and linking programs. You can do this in one of two ways:

1. Using the **–YSYSTEM_FIVE** option for the `cc` command.
2. Globally setting the environment variable PROG_ENV to SYSTEM_FIVE. If you are using the C shell, you would execute the following line, or include it in your .login file:

   ```
   setenv PROG_ENV SYSTEM_FIVE
   ```

   If you are using the Bourne shell, you would execute the following line, or include it in your .profile file:

   ```
   PROG_ENV=SYSTEM_FIVE ; export PROG_ENV
   ```

In both cases, the `cc`(1) command defines the preprocessor symbol SYSTEM_FIVE, so that the C preprocessor, `/lib/cpp`, will select the System V version of various data structures and symbol definitions.

In addition, if `cc`(1) invokes `ld`(1), the library libcV.a (the System V version of the Standard C library) is searched before libc.a to resolve references to the System-V-specific routines. Also, if **–lm** is specified on either the `cc`(1) or the `ld`(1) command line, then the System V version of the math library will be used instead of the regular ULTRIX math library.

The POSIX compatibility features are included in the library libcP.a, so the only special action needed is to specify **-YPOSIX** on the `cc`(1) command line or set the environment variable PROG_ENV to POSIX. Either action will cause the `cc`(1) command to define the preprocessor symbol POSIX and search the POSIX library.

Files

/usr/lib/libc.a	
/usr/lib/lib_cg.a	(VAX only)
/usr/lib/libm.a	
/usr/lib/libc_p.a	(VAX only)
/usr/lib/m_g.a	(VAX only)
/usr/lib/libm_p.a	(VAX only)

Diagnostics

Functions in the math library (3m) may return conventional values when the function is undefined for the given arguments or when the value is not representable. In these cases the external variable *errno* is set to the value EDOM (domain error) or ERANGE (range error). For further information, see `intro`(2). The values of EDOM and ERANGE are defined in the include file <math.h>.

See Also

cc(1), ld(1), nm(1), intro(2) intro(3), intro(3s), intro(3f), intro(3m), intro(3n)

Name

a64l, l64a – convert long integer and base-64 ASCII string

Syntax

long a64l (s)
char *s;

char *l64a (l)
long l;

Description

These functions are used to maintain numbers stored in base-64 ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a ''digit'' in a radix-64 notation.

The characters used to represent ''digits'' are **.** for 0, **/** for 1, **0** through **9** for 2–11, **A** through **Z** for 12–37, and **a** through **z** for 38–63.

The `a64l` subroutine takes a pointer to a null-terminated base-64 representation and returns a corresponding **long** value. If the string pointed to by *s* contains more than six characters, `a64l` will use the first six.

The `l64a` subroutine takes a **long** argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, `l64a` returns a pointer to a null string.

Restrictions

The value returned by `l64a` is a pointer into a static buffer, the contents of which are overwritten by each call.

Name

abort – generate an illegal instruction fault

Syntax

#include <stdlib.h>

void abort()

Description

The `abort` subroutine executes an instruction which is illegal in user mode. This causes a signal that normally terminates the process with a core dump, which may be used for debugging.

Diagnostics

Illegal instruction – core dumped
– Bourne shell.

Illegal instruction (core dumped)
– C shell.

Environment

When your program is compiled using the System V or POSIX environment, `abort` closes open files before aborting the process with an IOT fault.

Restrictions

The `abort` function does not flush standard I/O buffers. Use `fflush`(3s). For further information, see `fclose`(3s).

See Also

adb(1), exit(2), sigvec(2), fclose(3s)

Name

abs, labs – integer absolute value

Syntax

#include <stdlib.h>
#include <stdlib.h>

long labs(*i***)**
long *i***;**

int abs(*i***)**
int *i***;**

long labs(*i***)**
long *i***;**

Description

The `abs` and `labs` functions return the absolute value of their integer operand. The labs function does the same for a long int.

Restrictions

Applying the `abs` or `labs` function to the most negative integer generates a result which is the most negative integer. That is,

```
abs(0x80000000)
```

returns 0x80000000 as a result.

See Also

floor(3m)

Name

alarm – schedule signal after specified time

Syntax

#include <unistd.h>

unsigned alarm(*seconds*)
unsigned *seconds*;

Description

The `alarm` subroutine causes signal SIGALRM, see `signal`(3), to be sent to the invoking process in a number of seconds given by the argument. Unless caught or ignored, the signal terminates the process.

The `alarm` requests are not stacked. Successive calls reset the alarm clock. If the argument is 0, any `alarm` request is canceled. Because of scheduling delays, resumption of execution of when the signal is caught may be delayed an arbitrary amount. The longest specifiable delay time is 100000000 seconds. Values larger than 100000000 will be silently rounded down to 100000000.

The return value is the amount of time previously remaining in the alarm clock.

Environment

When your program is compiled using the System V environment, `alarm` rounds up any positive fraction of a second to the next second.

When your program is compiled using the POSIX environment, `alarm` takes a parameter of type unsigned, and returns a value of type unsigned.

See Also

getitimer(2), sigpause(2), sigvec(2), signal(3), sleep(3)

Name

assert – program verification

Syntax

#include <assert.h>

assert(*expression*)

Description

The `assert` macro indicates *expression* is expected to be true at this point in the program. It causes an `abort`(3) with a diagnostic comment on the standard error when *expression* is false (0). Compiling with the `cc`(1) option –DNDEBUG effectively deletes `assert` from the program.

Diagnostics

'Assertion failed: a, file *f n*'. The *a* is the assertion that failed; *f* is the source file and *n* the source line number of the `assert` statement.

Name

atof, atoi, atol, strtol, strtoul, strtod – convert ASCII to numbers

Syntax

#include <math.h>

double atof(*nptr***)**
char ****nptr*;**

atoi(*nptr***)**
char ****nptr*;**

long atol(*nptr***)**
char ****nptr*;**

long strtol(*nptr***, *eptr*, *base*)**
char ****nptr*, *****eptr***;**
int *base*;

unsigned long strtoul(*nptr***, *eptr*, *base*)**
char ****nptr*, *****eptr***;**
int *base*;

double strtod (*nptr***, *eptr*)**
char ****nptr*, *****eptr***;**

unsigned long strtoul(*nptr, eptr, base***)**
char ****nptr, *****eptr*;**
int *base*;

Description

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

The `atof` function recognizes (in order), an optional string of spaces, an optional sign, a string of digits optionally containing a radix character, an optional 'e' or 'E', and then an optionally signed integer.

The `atoi` and `atol` functions recognize (in order), an optional string of spaces, an optional sign, then a string of digits.

The `strtol` function returns as a long integer, the value represented by the character string *nstr*. The string is scanned up to the first character inconsistent with the *base*. Leading white-space characters are ignored.

If the value of *eptr* is not (char **) NULL, a pointer to the character terminating the scan is returned in ***eptr*. If no integer can be formed, ***eptr* is set to *nstr* , and zero is returned.

If *base* is positive and not greater than 36, it is used as the base for conversion. After an optional leading sign, leading zeros are ignored, and 0x or 0X is ignored if *base* is 16.

If *base* is zero, the string itself determines the base thus: After an optional leading sign, a leading zero indicates octal conversion, and a leading 0x or 0X hexadecimal conversion. Otherwise, decimal conversion is used.

Truncation from *long* to *int* can take place upon assignment, or by an explicit cast.

The `strtoul` function is the same as `strtol` except that `strtoul` returns, as an unsigned long integer, the value represented by the character string *nstr*.

The `strtod` function returns as a double-precision floating point number, the value represented by the character string pointed to by *nptr*. The string is scanned up to the first unrecognized character.

The `strtod` function recognizes an optional string of white-space characters, as defined by *isspace* in `ctype`, then an optional sign, then a string of digits optionally containing a radix character, then an optional *e* or *E* followed by an optional sign or space, followed by an integer.

If the value of *eptr* is not (char **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *eptr*. If no number can be formed, **eptr* is set to *nptr*, and zero is returned.

The radix character for `atof` and `strtod` is that defined by the last successful call to `setlocale` category `LC_NUMERIC`. If `setlocale` category `LC_NUMERIC` has not been called successfully, or if the radix character is not defined for a supported language, the radix character is defined as a period (.).

International Environment

LC_CTYPE If this environment variable is set and valid, `strtod` uses the international language database named in the definition to determine character classification rules.

LC_NUMERIC If this environment is set and valid, `atof` and `strtod` use the international language database named in the definition to determine radix character rules.

LANG If this environment variable is set and valid `atof` and `strtod` use the international language database named in the definition to determine collation and character classification rules. If `LC_CTYPE` or `LC_NUMERIC` is defined, their definition supercedes the definition of LANG.

Diagnostics

The `atof` function returns HUGE if an overflow occurs, and a 0 value if an underflow occurs, and sets *errno* to ERANGE. HUGE is defined in `<math.h>`.

The `atoi` function returns INT_MAX or INT_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `atol` function returns LONG_MAX or LONG_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtol` function returns LONG_MAX or LONG_MIN (according to the sign of the value) and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtoul` function returns ULONG_MAX and sets *errno* to ERANGE, if the correct value is outside the range of values that can be represented.

The `strtod` function returns HUGE (according to the sign of the value), and sets *errno* to ERANGE if the correct value would cause overflow. A 0 is returned and *errno* is set to ERANGE if the correct value would cause underflow.

See Also

ctype(3), setlocale(3), scanf(3s), environ(5int)

Name

bsearch – binary search a sorted table

Syntax

#include <stdlib.h>

void *bsearch (*key*, *base*, *nel*, **sizeof** (**key*), *compar*)
void **key*, **base*;
size_t *nel*;
int (**compar*)( );

Description

The `bsearch` subroutine is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating where a datum may be found. The table must be previously sorted in increasing order according to a provided comparison function. The *key* points to the datum to be sought in the table. The *base* points to the element at the base of the table. The *nel* is the number of elements in the table. The *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according to whether the first argument is to be considered less than, equal to, or greater than the second.

Diagnostics

A NULL pointer is returned if the key cannot be found in the table.

Notes

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

See Also

hsearch(3), lsearch(3), qsort(3), tsearch(3)

Name

bcopy, bcmp, bzero, ffs – bit and byte string operations

Syntax

bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b1, length)
char *b1;
int length;

ffs(i)
int i;

Description

The functions `bcopy`, `bcmp`, and `bzero` operate on variable length strings of bytes. They do not check for null bytes as the routines in `string`(3) do.

The `bcopy` function copies *length* bytes from string *b1* to the string *b2*.

The `bcmp` function compares byte string *b1* against byte string *b2*, returning zero if they are identical, non-zero otherwise. Both strings are assumed to be *length* bytes long.

The `bzero` function places *length* 0 bytes in the string *b1*.

The *ffs* finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1. A return value of 0 indicates the value passed is zero.

Restrictions

The `bcmp` and `bcopy` routines take parameters backwards from `strcmp` and `strcpy`.

Name

clock – report CPU time used

Syntax

#include <time.h>

clock_t clock ( )

CLOCKS_PER_SEC

Description

The `clock` routine returns the amount of CPU time (in microseconds) used since the first call to `clock`. The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed `wait`(2) or `system`(3). To determine the time in seconds, the value returned by `clock` should be divided by the value of the macro CLOCKS_PER_SEC.

The resolution of the clock is 16.667 milliseconds.

Restrictions

The value returned by `clock` is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned will wrap around after accumulating only 2147 seconds of CPU time (about 36 minutes).

See Also

wait(2), times(3), system(3)

Name

toupper, tolower, _toupper, _tolower, toascii – translate characters

Syntax

#include <ctype.h>

int toupper(*c***)**
int c;

int tolower(*c***)**
int c;

int _toupper(*c***)**
int c;

int _tolower(*c***)**
int c;

int toascii(*c***)**
int c;

Description

The functions `toupper` and `tolower` have as their domain the range of the `getc` function. If the argument to `toupper` represents a lowercase letter, the output from the fuction is the corresponding uppercase letter. If the argument to `tolower` represents an uppercase letter, the result is the corresponding lowercase letter.

The case of *c* depends on the definition of the character in the language database. Because the case of a character can vary between language databases, the case of *c* depends on what language database is in use. Specifically, the case of arguments depends on what property tables are associated the LC_CTYPE category. Property tables are associated with the LC_CTYPE category by a successful call to the `setlocale` function that includes the LC_CTYPE category. If no successful call to define LC_CTYPE has occurred or if the character case information is unavailable for the language in use, the rules of the ASCII coded character set determine the case of arguments.

If the argument to the `toupper` function does not have the uppercase attribute, `toupper` returns the argument unchanged. Likewise, if the argument to the `tolower` function does not have the lowercase attribute, `tolower` returns it unchanged.

The macros `_toupper` and `_tolower` have the same affect as `toupper` and `tolower`. The difference is that the argument to the macros must be an ASCII character (that is, a character in the domain –1 to 127) and the argument must have the appropriate case. Arguments to `_toupper` must have the uppercase attribute and arguments to `_tolower` must the lowercase attribute. The result of supplying arguments to these macros that are outside the domain or do not have the appropriate case is undefined. These macros operate faster than the `toupper` and `tolower` functions.

The macro `toascii` converts its argument to the ASCII character set. The macro converts its argument by truncating the numerical representation of the argument so that it is between –1 and 127. You can use this macro when you move an application

to a system other than an ULTRIX system.

International Environment

LC_CTYPE If this environment variable is set and valid, `conv` uses the international language database named in the definition to determine character classification rules.

See Also

ctype(3int), setlocale(3), getc(3)

crypt(3)

Name

crypt, crypt16, setkey, encrypt – DES encryption

Syntax

char *crypt(*key, salt*)
char **key*, **salt*;

char *crypt16(*key, salt*)
char **key*, **salt*;

setkey(*key*)
char **key*;

Description

The `crypt` subroutine is the password encryption routine. It is based on the NBS Data Encryption Standard, with variations intended to frustrate use of hardware implementations of the DES for key search.

The first argument to `crypt` is normally a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]. The *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the salt. The first two characters are the salt itself.

The `crypt16` subroutine is identical to the `crypt` function except that it will accept a password up to sixteen characters in length. It generates a longer encrypted password for use with enhanced security features.

The other entries provide primitive access to the actual DES algorithm. The argument of `setkey` is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the `encrypt` entry is likewise a character array of length 64 containing 0s and 1s. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the *key* set by `setkey`. If *edflag* is 0, the argument is encrypted; if non-zero, it is decrypted.

Restrictions

The return values from `crypt` and `crypt16` point to static data areas whose content is overwritten by each call.

Environment

Default Environment

In the default environment on systems that do not have the optional encryption software installed the `encrypt` function expects exactly one argument, the data to be encrypted. The *edflag* argument is not supplied and there is no way to decrypt data. If the optional encryption software is installed the `encrypt` function behaves

as it does in the POSIX environment. The syntax for the default environment follows:

encrypt(*block*)
char ***block**;

POSIX Environment

In the POSIX environment the encrypt function always expects two arguments. The `encrypt` function will set *errno* to ENOSYS and return if *edflag* is non-zero and the optional encryption software is not present. The syntax for the POSIX environment follows:

encrypt(*block, edflag*)
char ***block**;
int *edflag*;

In all cases the `setkey` function will set *errno* to ENOSYS and return if the optional encryption software is not present.

See Also

login(1), passwd(1), yppasswd(1yp), getpass(3), auth(5), passwd(5), passwd(5yp)
ULTRIX Security Guide for Users and Programmers

Name

ctime, localtime, gmtime, asctime, difftime, mktime, timezone, tzset – date and time functions

Syntax

As shown, the `ctime`, `localtime`, `gmtime`, `asctime`, `difftime`, `mktime`, and `tzset` calls are common to both the non-System V environment and the System V environment.

Common to Both Environments

#include <time.h>

void tzset()

char *ctime(*clock***)**
time_t *****clock***;**

char *asctime(*tm***)**
struct *tm* ***tm*****;**

struct *tm* ***localtime(***clock***)**
time_t *****clock***;**

struct *tm* ***gmtime(***clock***)**
time_t *****clock***;**

double difftime(*time1***, *time0*)**
time_t *time1*, *time0*;

time_t mktime(*timeptr***)**
struct *tm* ***timeptr*****;**

extern char *tzname[2];

BSD Environment Only

char *timezone(*zone***, *dst*)**

System V and POSIX Environments Only

extern long timezone;

extern int daylight;

Description

The `tzset` call uses the value of the environment variable TZ to set up the time conversion information used by `localtime`.

If TZ does not appear in the environment, the file `/etc/zoneinfo/localtime` is used by `localtime`. If this file fails for any reason, the Greenwich Mean Time (GMT) offset as provided by the kernel is used. In this case, Daylight Savings Time (DST) is ignored, resulting in the time being incorrect by some amount if DST is currently in effect. If this fails for any reason, GMT is used.

If TZ appears in the environment but its value is a null string, GMT is used; if TZ appears and its value is not a null string, its value is interpreted using rules specific to the System V and non-System V environments.

Programs that always wish to use local wall clock time should explicitly remove the environmental variable TZ with `unsetenv(3)`.

The `ctime` call converts a long integer, pointed to by *clock,* representing the time in seconds since 00:00:00 GMT, January 1, 1970, and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Sun Sep 16 01:03:52 1985\n\0
```

The `localtime` and `gmtime` calls return pointers to *tm* structures, described below. The `localtime` call corrects for the time zone and possible DST; `gmtime` converts directly to GMT, which is the time the ULTRIX system uses.

The `asctime` call converts a *tm* structure to a 26-character string, as shown in the previous example, and returns a pointer to the string.

Declarations of all the functions and externals, and the *tm* structure, are in the <time.h> header file. The structure declaration is:

```
struct tm {
        int tm_sec;     /* seconds (0 - 59) */
        int tm_min;     /* minutes (0 - 59) */
        int tm_hour;    /* hours (0 - 23) */
        int tm_mday;    /* day of month (1 - 31) */
        int tm_mon;     /* month of year (0 - 11) */
        int tm_year;    /* year - 1900 */
        int tm_wday;    /* day of week (Sunday = 0) */
        int tm_yday;    /* day of year (0 - 365) */
        int tm_isdst;   /* flag: daylight savings time in
                           effect */
        long tm_gmtoff; /* offset from GMT in seconds */
        char *tm_zone;  /* abbreviation of timezone name */

};
```

tm_isdst is nonzero if DST is in effect.

tm_gmtoff is the offset (in seconds) of the time represented from GMT, with positive values indicating East of Greenwich.

The `difftime` call computes the difference between two calendar times: *time1* - *time0* and returns the difference expressed in seconds.

The `mktime` call converts the broken-down local time in the *tm* structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by `time`. The values of **tm_wday** and **tm_yday** in the structure are ignored, and the other values are not restricted to the ranges indicated above for the *tm* structure. A positive or zero value for **tm_isdst** causes `mktime` to presume that DST, respectively, is or is not in effect for the specified time. A negative value causes `mktime` to attempt to determine whether DST is in effect for the specified time. On successful completion, the values of **tm_wday** and **tm_yday** are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above. If the calendar time cannot be represented, the function returns the value **(time_t)–1**.

The external variable *tzname*, contains the current time zone names. The function `tzset` sets this variable.

BSD and POSIX Environment Only

If TZ appears in the environment and its value is not a null string, its value has one of three formats:

:

or

:*pathname*

or

stdoffset[dst[offset][,start[/time],end[/time]]]

If TZ is the single colon format (first format), GMT is used.

If TZ is the colon followed by a pathname format (second), the characters following the colon specify a pathname of a `tzfile`(5) format file from which to read the time conversion information. If the pathname begins with a slash, it represents an absolute pathname; otherwise the pathname is relative to the system time conversion information directory `/etc/zoneinfo`. If this file fails for any reason, the GMT offset as provided by the kernel is used.

If the first character in TZ is not a colon (third format), the components of the string have the following meaning:

std and *dst*
: Three or more characters that are the designation for the standard (*std*) or summer (*dst*) time zone. Only *std* is required; if *dst* is missing, then summer time does not apply in this locale. Upper- and lowercase letters are explicitly allowed. Any characters except a leading colon (:), digits, comma (,), minus (–), plus (+), and ASCII NUL are allowed.

offset
: Indicates the value to be added to the local time to arrive at Coordinated Universal Time. The *offset* has the form:

hh[:mm[:ss]]

The minutes (*mm*) and seconds (*ss*) are optional. The hour (*hh*) is required and may be a single digit. The *offset* following *std* is required. If no *offset* follows *dst*, summer time is assumed to be one hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between zero and 24, and the minutes (and seconds) – if present – between zero and 59. If preceded by a "–", the time zone is east of the Prime Meridian; otherwise it is west (which may be indicated by an optional preceding "+").

start and *end*
: Indicates when to change to and back from summer time. *Start* describes the date when the change from standard to summer time occurs and *end* describes the date when the change back happens. The format of *start* and *end* must be one of the following:

J*n*
: The Julian day *n* ($1 \leq n \leq 365$). Leap days are not counted. That is, in all years, including leap years,

February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n — The zero-based Julian day ($0 \le n \le 365$). Leap days are counted, and it is possible to refer to February 29.

M*m.n.d* — The *n*th *d* day of month *m* ($1 \le n \le 5$, $0 \le d \le 6$, $1 \le m \le 12$). When *n* is 5 it refers to the last *d* day of month *m*. Day 0 is Sunday.

time — The *time* field describes the time when, in current time, the change to or from summer time occurs. *Time* has the same format as *offset* except that no leading sign (a minus sign (–) or a plus sign (+)) is allowed. The default, if *time* is not given, is 02:00:00.

As an example of the previous format, if the TZ environment variable had the value EST5EDT4,M4.1.0,M10.5.0 it would describe the rule, which went into effect in 1987, for the Eastern time zone in the USA. Specifically, EST would be the designation for standard time, which is 5 hours behind GMT. EDT would be the designation for DST, which is 4 hours behind GMT. DST starts on the first Sunday in April and ends on the last Sunday in October. In both cases, since the time was not specified, the change to and from DST would occur at the default time of 2:00 AM.

The `timezone` call remains for compatibility reasons only; it is impossible to reliably map timezone's arguments (*zone*, a 'minutes west of GMT' value and *dst*, a 'daylight saving time in effect' flag) to a time zone abbreviation.

If the environmental string TZNAME exists, `timezone` returns its value, unless it consists of two comma separated strings, in which case the second string is returned if *dst* is non-zero, else the first string. If TZNAME does not exist, *zone* is checked for equality with a built-in table of values, in which case `timezone` returns the time zone or daylight time zone abbreviation associated with that value. If the requested *zone* does not appear in the table, the difference from GMT is returned; that is, in Afghanistan, `timezone(-(60*4+30),0)` is appropriate because it is 4:30 ahead of GMT, and the string 'GMT+4:30' is returned. Programs that in the past used the `timezone` function should return the *zone* name as set by `localtime` to assure correctness.

System V Environment Only

If TZ appears in the environment its value specifies a pathname of a `tzfile`(5) format file from which to read the time conversion information. If the pathname begins with a slash, it represents an absolute pathname; otherwise the pathname is relative to the system time conversion information directory `/etc/zoneinfo`.

If TZ appears in the environment and using the value as a pathname of a `tzfile`(5) format file fails for any reason, the value is assumed to be a three-letter time zone name followed by a number representing the difference between local time and GMT in hours, followed by an optional three-letter name for a time zone on DST. For example, the setting for New Jersey would be EST5EDT.

System V and POSIX Environment Only

The external *long* variable `timezone` contains the difference, in seconds, between GMT and local standard time (in EST, `timezone` is 5*60*60), The external variable *daylight* is nonzero if and only if the standard USA DST conversion should be applied. These variables are set whenever `tzset, ctime, localtime, mktime,` or `strftime` are called.

Restrictions

The return values point to static data whose content is overwritten by each call. The **tm_zone** field of a returned **struct tm** points to a static array of characters, which will also be overwritten at the next call (and by calls to `tzset`).

Files

/etc/zoneinfo	time zone information directory
/etc/zoneinfo/localtime	local time zone file

See Also

gettimeofday(2), getenv(3), strftime(3), time(3), tzfile(5), environ(7)

Name

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – character classification macros

Syntax

#include <ctype.h>

int isalpha (*c*)

int c;

Description

These macros classify character-coded integer values according to the rules of the coded character set (codeset) identified by the last successful call to `setlocale` category `LC_CTYPE`. All macros return non-zero for true and zero for false.

If `setlocale` category `LC_CTYPE` has not been called successfully, or if character classification information is not available for a supported language, then characters are classified according to the rules of the ASCII 7-bit coded character set, returning 0 for values above octal 0177.

The macro `isascii` provides a result for all integer values. The rest provide a result for EOF and values in the character range of the codeset identified by the last successful call to `setlocale` category `LC_CTYPE`.

`isalpha`	*c* is a letter
`isupper`	*c* is an uppercase letter
`islower`	*c* is a lowercase letter
`isdigit`	*c* is a digit
`isxdigit`	*c* is a hexadecimal digit, by default [0-9], [A-F], or [a-f]
`isalnum`	*c* is an alphanumeric character
`isspace`	*c* is a space, tab, carriage return, new line, or form feed
`ispunct`	*c* is a punctuation character (neither control, alphanumeric, nor space)
`isprint`	*c* is a printing character, by default code 040(8) (space) through 0176 (tilde)
`isgraph`	*c* is a printing character, like `isprint` except false for space
`iscntrl`	*c* is a delete character (0177) or ordinary control character (less than 040) except for space characters
`isascii`	*c* is an ASCII character, code less than 0200

International Environment

LC_CTYPE	If this environment variable is set and valid, `ctype` uses the international language database named in the definition to determine character classification rules.

LANG If this environment variable is set and valid, `ctype` uses the international language database named in the definition to determine the character classification rules. If `LC_CTYPE` is defined, that definition supercedes the definition of `LANG`.

See Also

conv(3), setlocale(3), stdio(3s), environ(5int), ascii(7)
Guide to Developing International Software

Name

opendir, readdir, telldir, seekdir, rewinddir, closedir – directory operations

Syntax

#include <sys/types.h>
#include <sys/dir.h>

DIR *opendir(*dirname*)
**char ***dirname*;

struct direct *readdir(*dirp*)
**DIR ***dirp*;

long telldir(*dirp*)
**DIR ***dirp*;

seekdir(*dirp, loc*)
**DIR ***dirp*;
long loc;

rewinddir(*dirp*)
**DIR ***dirp*;

int closedir(*dirp*)
**DIR ***dirp*;

Description

The `opendir` library routine opens the directory named by *filename* and associates a directory stream with it. A pointer is returned to identify the directory stream in subsequent operations. The pointer NULL is returned if the specified *filename* can not be accessed, or if insufficient memory is available to open the directory file.

The `readdir` routine returns a pointer to the next directory entry. It returns NULL upon reaching the end of the directory or on detecting an invalid `seekdir` operation. The `readdir` routine uses the `getdirentries` system call to read directories. Since the `readdir` routine returns NULL upon reaching the end of the directory or on detecting an error, an application which wishes to detect the difference must set errno to 0 prior to calling `readdir`.

The `telldir` routine returns the current location associated with the named directory stream. Values returned by `telldir` are good only for the lifetime of the DIR pointer from which they are derived. If the directory is closed and then reopened, the `telldir` value may be invalidated due to undetected directory compaction.

The `seekdir` routine sets the position of the next `readdir` operation on the directory stream. Only values returned by `telldir` should be used with `seekdir`.

The `rewinddir` routine resets the position of the named directory stream to the beginning of the directory.

The `closedir` routine closes the named directory stream and returns a value of 0 if successful. Otherwise, a value of –1 is returned and errno is set to indicate the error. All resources associated with this directory stream are released.

Examples

The following sample code searches a directory for the entry *name*.

```
len = strlen(name);

dirp = opendir(".");

for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))

if (dp->d_namlen == len && !strcmp(dp->d_name, name)) {

            closedir(dirp);

            return FOUND;

      }

closedir(dirp);

return NOT_FOUND;
```

Environment

In the POSIX environment, the file descriptor returned in the DIR structure after an `opendir()` call will have the FD_CLOEXEC flag set. See **<fcntl.h>** for more detail.

Return Value

Upon successful completion, `opendir()` returns a pointer to an object of type DIR. Otherwise, a value of NULL is returned and errno is set to indicate the error.

The `readdir()` routine returns a pointer to an object of type struct dirent upon successful completion. Otherwise, a value of NULL is returned and errno is set to indicate the error. When the end of the directory is encountered, a value of NULL is returned and errno is not changed.

The `telldir()` routine returns the current location. No errors are defined for `telldir()`, `seekdir()`, and `rewinddir()`.

The `closedir()` routine returns zero upon successful completion. Otherwise, a value of -1 is returned and errno is set to indicate the error.

Diagnostics

The `closedir()` routine will fail if:

[EBADF] The *dirp* argument does not refer to an open directory stream.

[EINTR] The routine was interrupted by a signal.

The `opendir()` routine will fail if:

[EACCES] Search permission is denied for any component of *dirname* or read permission is denied for *dirname*.

[ENAMETOOLONG]
The length of the *dirname* string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}.

[ENOENT]	The *dirname* argument points to the name of a file which does not exist, or to an empty string and the environment defined is POSIX or SYSTEM_FIVE.
[ENOTDIR]	A component of *dirname* is not a directory.
[EMFILE]	Too many file descriptors are currently open for the process.
[ENFILE]	Too many files are currently open in the system.

The `readdir()` routine will fail if:

[EBADF]	The *dirp* argument does not refer to an open directory stream.

See Also

close(2), getdirentries(2), lseek(2), open(2), read(2), dir(5)

Name

div, ldiv – integer division

Syntax

#include <stdlib.h>

div_t div(*numer, denom***)**
int *numer***;**
int *denom***;**

ldiv_t ldiv(*numer, denom***)**
long *numer***;**
long *denom***;**

Description

The `div` and `ldiv` functions return the quotient and remainder of the division of the numerator *numer* by the denominator *denom*.

The return types div_t and ldiv_t are defined, in stdlib.h, as follows:

```
typedef struct {
            int    quot;  /* quotient */
            int    rem;   /* remainder */
      }     div_t;        /* result of div() */

typedef struct {
            long   quot;  /* quotient */
            long   rem;   /* remainder */
      }     ldiv_t;             /* result of ldiv() */
```

Restrictions

If division by zero is attempted, the behavior of `div` and `ldiv` is undefined.

Name

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

Syntax

double drand48 ( )

double erand48 (xsubi)
unsigned short xsubi[3];

long lrand48 ( )

long nrand48 (xsubi)
unsigned short xsubi[3];

long mrand48 ( )

long jrand48 (xsubi)
unsigned short xsubi[3];

void srand48 (seedval)
long seedval;

unsigned short *seed48 (seed16v)
unsigned short seed16v[3];

void lcong48 (param)
unsigned short param[7];

Description

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions `drand48` and `erand48` return non-negative double-precision floating-point values uniformly distributed over the interval [0.0, 1.0).

Functions `lrand48` and `nrand48` return non-negative long integers uniformly distributed over the interval $[0, 2^{31})$.

Functions `mrand48` and `jrand48` return signed long integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions `srand48, seed48` and `lcong48` are initialization entry points, one of which should be invoked before either `drand48, lrand48` or `mrand48` is called. Although it is not recommended practice, constant default initializer values will be supplied automatically if `drand48, lrand48` or `mrand48` is called without a prior call to an initialization entry point. Functions `erand48, nrand48` and `jrand48` do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c)_{\text{mod } m} \qquad n \geq 0.$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless `lcong48` has been invoked, the multiplier value a and the addend value c are given by

$a = 5DEECE66D_{16} = 273673163155_8$
$c = B_{16} = 13_8$.

The value returned by any of the functions `drand48, erand48, lrand48, nrand48, mrand48` or `jrand48` is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions `drand48, lrand48` and `mrand48` store the last 48-bit X_i generated in an internal buffer; that is why they must be initialized prior to being invoked. The functions `erand48, nrand48` and `jrand48` require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. That is why these routines do not have to be initialized. The calling program merely has to place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions `erand48, nrand48` and `jrand48` allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers. That is, the sequence of numbers in each stream will not depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function `srand48` sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer function `seed48` sets the value of X_i to the 48-bit value specified in the argument array. In addition, the previous value of X_i is copied into a 48-bit internal buffer, used only by `seed48`, and a pointer to this buffer is the value returned by `seed48`. This returned pointer, which can just be ignored if not needed, is useful if a program is to be restarted from a given point at some future time — use the pointer to get at and store the last X_i value, and then use this value to reinitialize via `seed48` when the program is restarted.

The initialization function `lcong48` allows the user to specify the initial X_i, the multiplier value a, and the addend value c. Argument array elements *param[0-2]* specify X_i, *param[3-5]* specify the multiplier a, and *param[6]* specifies the 16-bit addend c. After `lcong48` has been called, a subsequent call to either `srand48` or `seed48` will restore the ''standard'' multiplier and addend values, a and c, specified on the previous page.

Notes

The source code for the portable version can even be used on computers which do not have floating-point arithmetic. In such a situation, functions `drand48` and `erand48` do not exist. Instead, they are replaced by the two new functions below.

long irand48 (m)
unsigned short m;

long krand48 (xsubi, m)
unsigned short xsubi[3], m;

Functions `irand48` and `krand48` return non-negative long integers uniformly distributed over the interval [0, $m-1$].

See Also

rand(3)
ULTRIX Programmer's Manual, Unsupported

Name

ecvt, fcvt, gcvt – output conversion

Syntax

char *ecvt(*value, ndigit, decpt, sign*)
double value;
int *ndigit, *decpt, *sign*;

char *fcvt(*value, ndigit, decpt, sign*)
double value;
int *ndigit, *decpt, *sign*;

char *gcvt(*value, ndigit, buf*)
double value;
char **buf*;

Description

The `ecvt` routine converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer thereto. The position of the radix character relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is non-zero, otherwise it is zero. The low-order digit is rounded.

The `fcvt` routine is identical to `ecvt`, except that the correct digit has been rounded for FORTRAN F-format output of the number of digits specified by *ndigits*.

The `gcvt` routine converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F format if possible, otherwise E format is used, ready for printing. Trailing zeros may be suppressed.

The symbol used to represent a radix character is obtained from the last successful call to `setlocale` category `LC_NUMERIC`. The symbol can be determined by calling:

```
nl_langinfo (RADIXCHAR);
```

If `setlocale` category `LC_NUMERIC` has not been called successfully, or if the radix character is not defined for a supported language, the radix character defaults to a period (.).

International Environment

LC_NUMERIC If this environment is set and valid, `ecvt` uses the international language database named in the definition to determine radix character rules.

LANG If this environment is set and valid, `ecvt` uses the international language database named in the definition to determine radix character rules. If `LC_NUMERIC` is defined, its definition supercedes the definition of `LANG`.

Restrictions

The return values point to static data whose content is overwritten by each call.

See Also

setlocale(3), nl_langinfo(3int), printf(3int), printf(3s)
Guide to Developing International Software

emulate_branch (3)

Name

emulate_branch, execute_branch – branch emulation

Syntax

#include <signal.h>

emulate_branch(*scp, branch_instruction*)
struct sigcontext **scp*;
unsigned long *branch_instruction*;

execute_branch(*branch_instruction*)
unsigned long *branch_instruction*;

Description

The `emulate_branch` function is passed a signal context structure and a branch instruction. It emulates the branch based on the register values in the signal context structure. It modifies the value of the program counter in the signal context structure (*sc_pc*) to the target of the *branch_instruction*. The program counter must initially be pointing at the branch and the register values must be those at the time of the branch. If the branch is not taken the program counter is advanced to point to the instruction after the delay slot (*sc_pc* += 8).

If the branch instruction is a 'branch on coprocessor 2' or 'branch on coprocessor 3' instruction, `emulate_branch` calls `execute_branch` to execute the branch in data space to determine if it is taken or not.

Return Value

The `emulate_branch` function returns a 0 if the branch was emulated successfully. A non-zero value indicates the value passed as a branch instruction was not a branch instruction.

The `execute_branch` function returns non-zero on taken branches and zero on non-taken branches.

Restrictions

Since `execute_branch` is only intended to be used by `emulate_branch` it does not check its parameter to see if in fact it is a branch instruction. It is really a stop gap in case a coprocessor is added without the kernel fully supporting it (which is unlikely).

See Also

cacheflush(2), sigvec(2), signal(3)

Name

end, etext, edata – last locations in program

Syntax

```
extern end;
extern etext;
extern edata;
extern eprol;
```

Description

These names refer neither to routines nor to locations with interesting contents. The address of `etext` is the first address above the program text, `edata` above the initialized data region, and `eprol` is the first instruction of the user's program that follows the runtime startup routine.

When execution begins, the program break coincides with `end`, but it is reset by the routines `brk`(2), `malloc`(3), standard input/output `stdio`(3s), the profile (**–p**) option of `cc`(1), and so forth. The current value of the program break is reliably returned by sbrk(0). For further information, see `brk`(2).

See Also

cc(1), brk(2), malloc(3), stdio(3s)

Name

end, etext, edata – last locations in program

Syntax

extern end;
extern etext;
extern edata;

Description

These names refer neither to routines nor to locations with interesting contents. The address of `etext` is the first address above the program text, `edata` above the initialized data region, and `end` above the uninitialized data region.

When execution begins, the program break coincides with `end`, but it is reset by the routines `brk`(2), `malloc`(3), standard input/output `stdio`(3s), the profile (**–p**) option of `cc`(1), and so forth. The current value of the program break is reliably returned by 'sbrk(0)'. For further information, see `brk`(2).

See Also

brk(2), malloc(3), stdio(3s)

Name

execl, execv, execle, execlp, execvp, exect, environ – execute a file

Syntax

execl(name, arg0, arg1, ..., argn, (char *)0)
char *name, *arg0, *arg1, ..., *argn;

execv(name, argv)
char *name, *argv[];

execle(name, arg0, arg1, ..., argn, (char *)0, envp)
char *name, *arg0, *arg1, ..., *argn, *envp[];

execlp(file, arg0, arg1, ..., argn, (char *)0)
char *file, *arg0, *arg1, ..., *argn;

execvp(file,argv)
char *file, *argv[];

exect(name, argv, envp)
char *name, *argv[], *envp[];

extern char **environ;

Description

These routines provide various interfaces to the `execve` system call. Refer to `execve`(2) for a description of their properties; only brief descriptions are provided here.

In all their forms, these calls overlay the calling process with the named file, then transfer to the entry point of the core image of the file. There can be no return from a successful exec. The calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. `execl` is useful when a known file with known arguments is being called; the arguments to `execl` are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The `execv` version is useful when the number of arguments is unknown in advance. The arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

The `exect` version is used when the executed file is to be manipulated with `ptrace`(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state.

VAX-11

On VAX-11 machines, this is done by setting the trace bit in the process status longword.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

The *argv* is directly usable in another `execv` because *argv*[*argc*] is 0.

The *envp* is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an ''='', and a null-terminated value. The array of pointers is terminated by a null pointer. The shell `sh`(1) passes an environment entry for each global shell variable defined when the program is called. See `environ`(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell `environ`, which is used by `execv` and `execl` to pass the environment to any subprograms executed by the current program.

The `execlp` and `execvp` routines are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Restrictions

If `execvp` is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv[0]* and *argv[–1]* will be modified before return.

Diagnostics

If the file cannot be found, if it is not executable, if it does not start with a valid magic number if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is –1. For further information, see `a.out`(5). Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

Files

/bin/sh	Shell, invoked if command file found by execlp or execvp

See Also

csh(1), execve(2), fork(2), environ(7)

Name

execl, execv, execle, execlp, execvp, exect, environ – execute a file

Syntax

execl(*name, arg0, arg1, ..., argn, (char *)0*)
char **name, *arg0, *arg1, ..., *argn*;

execv(*name, argv*)
char **name, *argv[]*;

execle(*name, arg0, arg1, ..., argn, (char *)0, envp*)
char **name, *arg0, *arg1, ..., *argn, *envp[]*;

execlp(*file, arg0, arg1, ..., argn, (char *)0*)
char **file, *arg0, *arg1, ..., *argn*;

execvp(*file,argv*)
char **file, *argv[]*;

exect(*name, argv, envp*)
char **name, *argv[], *envp[]*;

extern *char **environ*;

Description

These routines provide various interfaces to the `execve` system call. Refer to `execve`(2) for a description of their properties; only brief descriptions are provided here.

In all their forms, these calls overlay the calling process with the named file, then transfer to the entry point of the core image of the file. There can be no return from a successful exec. The calling core image is lost.

The *name* argument is a pointer to the name of the file to be executed. The pointers *arg*[*0*], *arg*[*1*] ... address null-terminated strings. Conventionally *arg*[*0*] is the name of the file.

Two interfaces are available. `execl` is useful when a known file with known arguments is being called; the arguments to `execl` are the character strings constituting the file and the arguments; the first argument is conventionally the same as the file name (or its last component). A 0 argument must end the argument list.

The `execv` version is useful when the number of arguments is unknown in advance. The arguments to `execv` are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

The `exect` version is used when the executed file is to be manipulated with `ptrace`(2). The program is forced to single step a single instruction giving the parent an opportunity to manipulate its state. On the VAX-11 this is done by setting the trace bit in the process status longword.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

The *argv* is directly usable in another `execv` because *argv*[*argc*] is 0.

The *envp* is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an ''='', and a null-terminated value. The array of pointers is terminated by a null pointer. The shell `sh`(1) passes an environment entry for each global shell variable defined when the program is called. See `environ`(7) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell `environ`, which is used by `execv` and `execl` to pass the environment to any subprograms executed by the current program.

The `execlp` and `execvp` routines are called with the same arguments as `execl` and `execv`, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Restrictions

If `execvp` is called to execute a file that turns out to be a shell command file, and if it is impossible to execute the shell, the values of *argv[0]* and *argv[–1]* will be modified before return.

Diagnostics

If the file cannot be found, if it is not executable, if it does not start with a valid magic number, if maximum memory is exceeded, or if the arguments require too much space, a return constitutes the diagnostic; the return value is –1. For further information, see `a.out`(5). Even for the super-user, at least one of the execute-permission bits must be set for a file to be executed.

Files

/bin/sh	Shell, invoked if command file found by execlp or execvp

See Also

csh(1), execve(2), fork(2), environ(7)

Name

exit – terminate a process after flushing any pending output

Syntax

exit(*status*)
int *status*;
int atexit(*func*)
void (**func*)();

Description

The `exit` function terminates a process after calling the Standard I/O library function, *_cleanup,* to flush any buffered output. The `exit` function never returns.

The `atexit` function registers a function to be called (without arguments) at normal program termination; functions are called in the reverse order of their registration (that is, most recent first). If a function is registered more than once, it will be called more than once.

Return Value

The `atexit` function returns zero if the registration succeeds, or -1 if the function pointer is null or if too many functions are registered.

See Also

exit(2), intro(3s)

Name

fpc, get_fpc_csr, set_fpc_csr, swapRM, swapINX – floating-point control registers

Syntax

#include <mips/fpu.h>

int get_fpc_csr()

int set_fpc_csr(*csr***)**
int *csr***;**

int get_fpc_irr()

int swapRM(*x***)**
int *x***;**

int swapINX(*x***)**
int *x***;**

Description

These functions are to get and set the floating-point control registers of RISC floating-point units. All of these functions take and return their values as 32 bit integers.

The file **<mips/fpu.h>** contains unions for each of the control registers. Each union contains a structure that breaks out the bit fields into the logical parts for each control register. This file also contains constants for fields of the control registers.

RISC floating-point implementations have a control and status register and an implementation revision register. The control and status register is returned by `get_fpc_csr`. The routine `set_fpc_csr` sets the control and status register and returns the old value. The implementation revision register is read-only and is returned by the routine `get_fpc_irr`.

The function `swapRM` sets only the rounding mode and returns the old rounding mode. The function `swapINX` sets only the sticky inexact bit and returns the old one. The bits in the arguments and return values to `swapRM` and `swapINX` are right justified.

Name

fp_class – classes of IEEE floating-point values

Syntax

#include <fp_class.h>

int fp_class_d(double *x*);

int fp_class_f(float *x*);

Description

These routines are used to determine the class of IEEE floating-point values. They return one of the constants in the file <fp_class.h> and never cause an exception, even for signaling NaNs. These routines are to implement the recommended function class(x) in the appendix of the IEEE 754-1985 standard for binary floating-point arithmetic. The constants in <fp_class.h> refer to the following classes of values:

Constant	Class
FP_SNAN	Signaling NaN (Not-a-Number)
FP_QNAN	Quiet NaN (Not-a-Number)
FP_POS_INF	+∞ (positive infinity)
FP_NEG_INF	-∞ (negative infinity)
FP_POS_NORM	positive normalized nonzero
FP_NEG_NORM	negative normalized nonzero
FP_POS_DENORM	positive denormalized
FP_NEG_DENORM	negative denormalized
FP_POS_ZERO	+0.0 (positive zero)
FP_NEG_ZERO	-0.0 (negative zero)

Also See

ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic

frexp (3)

Name

frexp, ldexp, modf – split into mantissa and exponent

Syntax

#include <math.h>

double frexp(*value, eptr*)
double *value*;
int **eptr*;

double ldexp(*value, exp*)
double *value*;

double modf(*value, iptr*)
double *value*, **iptr*;

Description

The `frexp` subroutine returns the mantissa of a double *value* as a double quantity, *x*, of magnitude less than 1.0 and greater than or equal to 0.5 (0.5 <= |x| < 1) and stores an integer *n* such that *value* = *x*2**n* indirectly through *eptr*.

The `ldexp` returns the quantity *value*2**exp*.

The `modf` returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

Return Value

If `ldexp` would cause overflow, ±HUGE_VAL is returned (according to the sign of *value*) and *errno* is set to ERANGE. If `ldexp` would cause underflow, 0 is returned and *errno* is set to ERANGE.

Name

ftoi, itof, dtoi, itod, gtoi, itog – convert floating values between VAX and IEEE format

Syntax

int ftoi(*value*)
float **value*;

int itof(*value*)
float **value*;

int dtoi(*value*)
double **value*;

int itod(*value*)
double **value*;

int gtoi(*value*)
double **value*;

int itog(*value*)
double **value*;

Description

The following C library functions convert floating values between VAX and IEEE formats.

The `ftoi` function converts the specified VAX ffloat number to IEEE single-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itof` function converts the specified IEEE single-precision number to VAX ffloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, overflow).

The `dtoi` function converts the specified VAX dfloat number to IEEE double-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itod` function converts the specified IEEE double-precision number to VAX dfloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow or overflow).

The `gtoi` function converts the specified VAX gfloat number to IEEE double-precision format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

The `itog` function converts the specified IEEE double-precision number to VAX gfloat format. It returns zero if successful and nonzero without performing the conversion if not successful (for example, underflow).

ftok(3)

Name

ftok – standard interprocess communication package

Syntax

#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(path, id)
char *path;
char id;

Description

All interprocess communication facilities require the user to supply a key to be used by the `msgget`(2), `semget`(2), and `shmget`(2) system calls to obtain interprocess communication identifiers. One suggested method for forming a key is to use the `ftok`, file to key, subroutine described below. Another way to compose keys is to include the project ID in the most significant byte and to use the remaining portion as a sequence number. There are many other ways to form keys, but it is necessary for each system to define standards for forming them. If some standard is not adhered to, it will be possible for unrelated processes to unintentionally interfere with each other's operation. Therefore, it is strongly suggested that the most significant byte of a key in some sense refer to a project so that keys do not conflict across a given system.

The `ftok` subroutine returns a key based on *path* and *id* that is usable in subsequent `msgget, semget,` and `shmget` system calls. The *path* must be the path name of an existing file that is accessible to the process. The *id* is a character which uniquely identifies a project. Note that `ftok` will return the same key for linked files when called with the same *id* and that it will return different keys when called with the same file name but different *ids*.

Return Value

The `ftok` subroutine returns **(key_t) –1** if *path* does not exist or if it is not accessible to the process.

Warning

If the file whose *path* is passed to `ftok` is removed when keys still refer to the file, future calls to `ftok` with the same *path* and *id* will return an error. If the same file is recreated, then `ftok` is likely to return a different key than it did the original time it was called.

See Also

intro(2), msgget(2), semget(2), shmget(2)

Name

ftw – walk a file tree

Syntax

#include <ftw.h>

int ftw (*path, fn, depth*)
char ***path**;
int (***fn*) ( );
int *depth*;

Description

The `ftw` subroutine recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, `ftw` calls *fn*, passing it a pointer to a null-terminated character string containing the name of the object, a pointer to a **stat** structure containing information about the object, and an integer. For further information, see `stat`(2). Possible values of the integer, defined in the <ftw.h> header file, are FTW_F for a file, FTW_D for a directory, FTW_DNR for a directory that cannot be read, and FTW_NS for an object for which **stat** could not successfully be executed. If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the the contents of the **stat** structure will be undefined. An example of an object that would cause FTW_NS to be passed to *fn* would be a file in a directory with read but without execute (search) permission.

The `ftw` subroutine visits a directory before visiting any of its descendants.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some error is detected within `ftw` (such as an I/O error). If the tree is exhausted, `ftw` returns zero. If *fn* returns a nonzero value, `ftw` stops its tree traversal and returns whatever value was returned by *fn*. If `ftw` detects an error, it returns –1, and sets the error type in *errno*.

The `ftw` subroutine uses one file descriptor for each level in the tree. The *depth* argument limits the number of file descriptors so used. If *depth* is zero or negative, the effect is the same as if it were 1. The *depth* must not be greater than the number of file descriptors currently available for use. The `ftw` subroutine will run more quickly if *depth* is at least as large as the number of levels in the tree.

Restrictions

Because `ftw` is recursive, it is possible for it to terminate with a memory fault when applied to very deep file structures.
It could be made to run faster and use less storage on deep structures at the cost of considerable complexity.
The `ftw` subroutine uses `malloc`(3) to allocate dynamic storage during its operation. If `ftw` is forcibly terminated, such as by *longjmp* being executed by *fn* or an interrupt routine, `ftw` will not have a chance to free that storage, so it will remain permanently allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have *fn* return a nonzero value at its next invocation.

Diagnostics

[EACCES] Search permission is denied on a component of *path* or read permission is denied for path.

[ENAMETOOLONG] The length of the path string exceeds {PATH_MAX}, or a pathname component is longer than {NAME_MAX}.

[ENOENT] The path argument points to the name of a file which does not exist, or to an empty string and the environment defined is POSIX or SYSTEM_FIVE.

[ENOTDIR] A component of *path* is not a directory.

[ENOMEM] Not enough memory was available to complete the file tree walk.

See Also

stat(2), malloc(3)

Name

getauthuid, storeauthent, setauthfile, endauthent – get/set auth entry

Syntax

#include <sys/types.h>
#include <auth.h>

AUTHORIZATION *getauthuid(*uid***)**
uid_t *uid***;**

int storeauthent(*auth***)**
AUTHORIZATION ***auth*;**

void setauthfile(*pathname***)**
char ***pathname*;**

int endauthent()

Description

The `getauthuid` function looks up the auth entry for the specified user ID and returns a pointer to a static area containing it.

The `storeauthent` function will store the specified auth entry into the local auth database, overwriting any existing entry with the same *a_uid* field.

The `setauthfile` function will set the pathname of the file to be used for the local auth database in all subsequent operations.

The `endauthent` functions closes the auth database. Subsequent calls to getauthuid and storeauthent will reopen it.

The auth database may be distributed via the BIND/Hesiod naming service.

Restrictions

Only the super-user and members of the group *authread* may read information from the auth database.

Only the super-user may modify the auth database.

The auth databse may not be distributed via the Yellow Pages service.

Return Value

Functions which return a pointer value will return the null pointer (0) on EOF or error. Other functions will return zero (0) on success and a negative value on failure.

Files

/etc/auth.[pag,dir]

See Also

getpwent(3), auth(5), edauth(8)
Security Guide for Users and Programmers
Security Guide for Administrators
Guide to the BIND/Hesiod Service

Name

getcwd – get pathname of working directory

Syntax

char **getcwd (buf, size)*
char **buf*;
int *size*;

Description

The `getcwd` subroutine returns a pointer to the current directory pathname. The value of *size* must be at least two greater than the length of the pathname to be returned.

If *buf* is a NULL pointer, `getcwd` will obtain *size* bytes of space using `malloc`(3) In this case, the pointer returned by `getcwd` may be used as the argument in a subsequent call to *free*.

The function is implemented by using `popen`(3) to pipe the output of the `pwd`(1) command into the specified string space.

Examples

```
char *cwd, *getcwd();
.
.
.
if ((cwd = getcwd((char *)NULL, 64)) == NULL) {

     perror("pwd");
     exit(1);
}
printf("%s\n", cwd);
```

Return Value

Returns NULL with *errno* set if *size* is not large enough, or if an error occurs in a lower-level function.

Diagnostics

[EINVAL]	The size argument is zero or negative.
[ERANGE]	The size argument is greater than zero, but is smaller than the length of the pathname+1;
[EACCES]	Read or search permission is denied for a component of the pathname.
[ENOMEM]	Insufficient storage space is available.

See Also

pwd(1), malloc(3), popen(3)

Name

getenv, setenv, unsetenv – manipulate environment variables

Syntax

char *getenv(*name***)**
char ***name*;**

setenv(*name***, *value*, *overwrite*)**
char ***name*, *value*;**
int *overwrite*;

void unsetenv(*name***)**
char ***name*;**

Description

The `getenv` subroutine searches the environment list for a string of the form *name* = *value* and returns a pointer to the string *value* if such a string is present, otherwise `getenv` returns the value 0 (NULL). For further information, see `environ`(7).

The `setenv` subroutine searches the environment list in the same manner as `getenv`. If the string *name* is not found, a string of the form *name=value* is added to the environment. If it is found, and *overwrite* is non-zero, its value is changed to *value*. The `setenv` subroutine returns 0 on success and -1 on failure, where failure is caused by an inability to allocate space for the environment.

The `unsetenv` subroutine removes all occurrences of the string *name* from the environment. There is no library provision for completely removing the current environment. It is suggested that the following code be used to do so.

```
static char     *envinit[1];
extern char     **environ;
environ = envinit;
```

All of these routines permit, but do not require, a trailing equals sign (=) on *name* or a leading equals sign on *value*.

See Also

csh(1), sh(1), execve(2), putenv(3), environ(7)

getgrent(3)

Name

getgrent, getgrgid, getgrnam, setgrent, endgrent – get group entry

Syntax

#include <grp.h>

struct group *getgrent()

struct group *getgrgid(*gid***)**
gid_t *gid***;**

struct group *getgrnam(*name***)**
char ***name*;**

setgrent()

endgrent()

Description

The `getgrent`, `getgrgid` and `getgrnam` subroutines each return pointers to an object with the following structure containing the broken-out fields of a line in the group database:

```
struct  group { /* see getgrent(3) */
        char    *gr_name;
        char    *gr_passwd;
        int     gr_gid;
        char    **gr_mem;
};

struct group *getgrent(), *getgrgid(), *getgrnam();
```

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group-ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

A call to `setgrent` has the effect of rewinding the group file to allow repeated searches. The `endgrent` may be called to close the group database when processing is complete.

The `getgrent` subroutine simply reads the next line while `getgrgid` and `getgrnam` search until a matching *gid* or *name* is found (or until EOF is encountered). The `getgrent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setgrent` must be made before a `while` loop using `getgrent` in order to perform initialization and an `endgrent` must be used after the loop. Both `getgrgid` and `getgrnam` make calls to `setgrent` and `endgrent`.

Restrictions

All information is contained in a static area so it must be copied if it is to be saved.

If YP is running, `getgrent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The group database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

A null pointer (0) is returned on EOF or error.

Files

`/etc/group`

See Also

group(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

gethostsex(3)

Name

gethostsex – get the byte sex of the host machine

Syntax

#include <sex.h>
int gethostsex()

Description

The `gethostsex` routine returns one of two constants, BIGENDIAN or LITTLEENDIAN, for the sex of the host machine. These constants are in **sex.h.**

See Also

swapsex(3)

Name

getlogin – get login name

Syntax

char *getlogin()

Description

The `getlogin` subroutine returns a pointer to the login name as found in `/etc/utmp`. It may be used in conjunction with `getpwnam` to locate the correct password file entry when the same userid is shared by several login names.

If `getlogin` is called within a process that is not attached to a typewriter, it returns NULL. The correct procedure for determining the login name is to first call `getlogin` and if it fails, to call `getpw` (`getuid`).

Restrictions

The return values point to static data whose content is overwritten by each call.

Return Value

Returns NULL (0) if name not found.

Files

/etc/utmp

See Also

getgrent(3), getpw(3), getpwent(3), utmp(5)

getmountent(3)

Name

getmountent – get information about mounted file systems without blocking

Syntax

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/mount.h>

getmountent(start, buffer, nentries)
int             *start;
struct fs_data  *buffer;
int              nentries;
```

Description

The `getmountent` library routine retrieves mounted file system information from memory without blocking. The file system information retrieved (the number of free inodes and blocks) might not be up to date. If the accuracy of the file system information retrieved is critical, you should use `statfs` or `getmnt` instead of `getmountent`.

The *start* argument is the current logical location within the internal system mount table and must be initially set to 0. The *buffer* argument is the holding area for the returned information; that is, the `fs_data` structures. The size of *buffer* should be at least the number of entries times the size of the `fs_data` structure, in bytes.

The *nentries* argument defines the number of mount table entries that are to be retrieved.

The number of file systems described by the information placed in *buffer* is returned. The *start* argument is updated so that successive calls can be used to retrieve the entire mount table.

Return Value

Upon successful completion, a value indicating the number of `struct fs_data` structures stored in *buffer* is returned. If there are no more file systems in the mount table, `0` is returned. Otherwise, `-1` is returned and the global variable *errno* is set to indicate the error.

Diagnostics

EINVAL	Invalid argument.
EFAULT	Either *buffer* or *start* causes an illegal address to be referenced.
EIO	An I/O error occurred while reading from the file system.

See Also

getmnt(2), statfs(3)

Name

getopt – get option letter from argument vector

Syntax

```
#include <stdio.h>
int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;

extern char *optarg;
extern int optind, opterr;
```

Description

The `getopt` subroutine returns the next option letter in *argv* that matches a letter in *optstring*. The *optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by white space. The *optarg* is set to point to the start of the option argument on return from `getopt`.

The function `getopt` places in *optind* the *argv* index of the next argument to be processed. The external variable optind is automatically initialized to 1 before the first call to `getopt`.

When all options have been processed (that is, up to the first non-option argument), `getopt` returns EOF. The special option — may be used to delimit the end of the options; EOF will be returned, and — will be skipped.

Diagnostics

The function `getopt` prints an error message on *stderr* and returns a question mark (**?**) when it encounters an option letter that is not included in *optstring*. Setting opterr to 0 disables this error message.

Examples

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options **a** and **b**, and the options **f** and **o**, both of which require arguments:

```
#include <stdio.h>
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind, opterr;
        extern char *optarg;
        .
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
```

```
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (2);
        }
        for ( ; optind < argc; optind++) {
            if (access (argv[optind], 4)) {
        .
        .
        .
}
```

See Also

getopt(1)

Name

getpass – read a password

Syntax

char *getpass(prompt)
char *prompt;

Description

The `getpass` subroutine reads a password from the file `/dev/tty`, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. The `getpass` subroutine can return up to `PASS_MAX` characters. `PASS_MAX` is defined in `/usr/include/sys/limits.h`. A pointer is returned to a null-terminated string of at most 16 characters.

Environment

When your program is compiled using the System V environment, if the file `/dev/tty` cannot be opened, a NULL pointer is returned. An interrupt will terminate input and send an interrupt signal to the calling process before returning.

Restrictions

The return value points to static data whose content is overwritten by each call.

Files

`/dev/tty`

See Also

`crypt`(3)

Name

getpw – get name from uid

Syntax

getpw(uid, buf)
char *buf;

Description

The `getpw` routine has been superseded by `getpwuid`, see `getpwent`(3).

The `getpw` routine searches the password file for the (numerical) *uid*, and fills in *buf* with the corresponding line; it returns nonzero if *uid* could not be found. The line is null terminated.

Diagnostics

Nonzero return on error.

Files

/etc/passwd

See Also

getpwent(3), passwd(5yp)

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent, setpwfile – get password entry

Syntax

#include <pwd.h>

struct passwd *getpwent()

struct passwd *getpwuid(*uid***)**
uid_t *uid***;**

struct passwd *getpwnam(*name***)**
char ***name*;**

void setpwent()

void endpwent()

void setpwfile(*pathname***)**
**char ***pathname*

Description

The routines, `getpwent`, `getpwuid` and `getpwnam`, each return a pointer to an object with the following structure containing the broken-out fields of a line in the password database:

```
struct  passwd { /* see getpwent(3) */
        char    *pw_name;
        char    *pw_passwd;
        uid_t    pw_uid;
        gid_t    pw_gid;
        int      pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};

struct passwd *getpwent(), *getpwuid(), *getpwnam();
```

The fields *pw_quota* and *pw_comment* are unused; the others have meanings described in `passwd`(5).

A call to `setpwent` has the effect of rewinding the password file to allow repeated searches. `Endpwent` may be called to close the password database when processing is complete.

The `getpwent` subroutine simply retieves the next entry while `getpwuid` and `getpwnam` search until a matching *uid* or *name* is found (or until all entries are exhausted). The `getpwent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire database.

A call to `setpwent` must be made before a `while` loop using `getpwent` in order to perform initialization and an `endpwent` must be used after the loop. Both `getpwuid` and `getpwnam` make calls to `setpwent` and `endpwent`.

The `setpwfile` subroutine sets the pathname of the ASCII passwd file and optional hashed database to be used for local passwd lookups. If a passwd file has been left open by a call to `setpwent` or `getpwent`, `setpwfile` will close it first. `Setpwfile` does not directly affect the use of distributed passwd databases.

Restrictions

All information is contained in a static area so it must be copied if it is to be saved.

If YP is running, `getpwent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The password database may also be distributed via the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

Null pointer (0) returned on EOF or error.

Files

`/etc/passwd`

See Also

getlogin(3), passwd(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getrpcent, getrpcbynumber, getrpcbyname, setrpcent, endrpcent – get rpc entry

Syntax

#include <netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbynumber(*number***)**
int *number***;**

struct rpcent *getrpcbyname(*name***)**
char ***name*;**

setrpcent(*stayopen***)**
int *stayopen***;**

endrpcent( )

Description

The `getrpcent`, `getrpcbynumber` and `getrpcbyname` subroutines each return pointers to an object with the following structure containing the broken-out fields of a line in the rpc database:

```
struct  rpcent {                /* see getrpcent(3) */
        char    *r_name;
        char    **r_aliases;    /* alias list */
        char    r_number;       /* rpc program number */
};
struct group *getrpcent(), *getrpcbynumber(), *getrpcbyname();
```

The members of this structure are:

r_name The name of the rpc.
r_aliases A zero-terminated list of alternate names for the rpc.
r_number The rpc program number for the rpc.

If the *stayopen* flag on the `setrpcent` subroutine is NULL, the `rpc` database is opened. Otherwise the `setrpcent` has the effect of rewinding the `rpc` database. The `endrpcent` may be called to close the `rpc` file when processing is complete.

The `getrpcent` subroutine simply reads the next line while `getrpcbynumber` and `getrpcbyname` search until a matching *gid* or *name* is found (or until EOF is encountered). The `getrpcent` subroutine keeps a pointer in the database, allowing successive calls to be used to search the entire file.

A call to `setrpcent` must be made before a `while` loop using `getrpcent` in order to perform initialization and an `endrpcent` must be used after the loop. Both `getrpcbynumber` and `getrpcbyname` make calls to `setrpcent` and `endrpcent`.

Restrictions

All information is contained in a static area so it must be copied if it is to be saved.

If YP is running, `getrpcent` does not return the entries in any particular order. See the *Guide to the Yellow Pages Service* for setup information.

The `rpc` database may also be distributed by the BIND/Hesiod naming service. See the *Guide to the BIND/Hesiod Service* for more information.

Return Value

A null pointer (0) is returned on EOF or error.

Files

`/etc/rpc`

See Also

rpc(5), svc.conf(5)
Guide to the BIND/Hesiod Service
Guide to the Yellow Pages Service

Name

getsvc - get a pointer to the svcinfo structure

Syntax

#include <sys/svcinfo.h>

struct svcinfo *getsvc()

Description

The `getsvc` call retrieves information from the system about the `svcinfo` structure by returning a pointer to the structure. This structure is initialized the first time a `getsvc` call is made. The contents of the `/etc/svc.conf` file are parsed and stored in the `svcinfo` structure. If the `/etc/svc.conf` file is modified, the contents of this structure will be updated upon the next `getsvc` call.

The `/etc/svc.conf` file contains the names of the databases that can be served by YP, BIND, or local files and the name service selection for each database. It also has settings for four security parameters. The database service selection and security parameters are stored in the `svcinfo` structure.

The following structure exists in the `svcinfo.h` file:

```
#define SVC_DATABASES 20
#define SVC_PATHSIZE 8
struct svcinfo {
    int svcdate;       /* Last mod date of /etc/svc.conf */

    int svcpath[SVC_DATABASES][SVC_PATHSIZE];    /* indexed by
                          databases and choice 0=first choice
                        1=second choice, etc value stored is
                      source */

       struct {
           int passlenmin;
           int passlenmax;
           int softexp;
           int seclevel;
       } svcauth;
};
```

The `svcdate` field contains the date that the `/etc/svc.conf` file was last modified. The `svcpath` array contains the name service choices for each database. The `svcauth` structure contains the values for the four security parameters: password length minimum (*passlenmin*), password length maximum (*passlenmax*), soft expiration date of a password (*softexp*), and security mode of a system (*seclevel*).

Examples

The following programming example shows how to use the `getsvc` call to use the information in the `svcinfo` structure to process specific host information.

```
#include <sys/svcinfo.h>
struct svcinfo *svcinfo;

if ((svcinfo = getsvc()) != NULL)
  for (i=0; (j = svcinfo->svcpath[SVC_HOSTS][i]) != SVC_LAST; i++)
        switch(j) {
            case SVC_BIND:
                /* process BIND hosts */
            case SVC_YP:
                /* process YP hosts */
            case SVC_LOCAL:
               /* process LOCAL hosts */
        }
```

Files

```
/etc/svc.conf
/usr/include/sys/svcinfo.h
```

See Also

svc.conf(5), svcsetup(8)

Name

getttyent, getttynam, setttyent, endttyent – get ttys file entry

Syntax

#include <ttyent.h>
struct ttyent *getttyent( )
struct ttyent *getttynam(*name* **) char** ***name;***
int setttyent( )
int endttyent( )

Description

These functions allow a program to access data in the file /etc/ttys. The `getttyent` function reads the `/etc/ttys` file line by line, opening the file if necessary. `setttyent` rewinds the file, and `endttyent` closes it. `getttynam` searches from the beginning of the file until a matching name is found, or until end-of-file is encountered.

The functions `getttyent` and `getttynam` each return a pointer to an object that has the following structure. Each element of the structure contains one field of a line in the `/etc/ttys` file.

```
struct ttyent {         /* see getttyent(3) */
       char *ty_name;   /* terminal device name */
       char *ty_getty;  /* command to execute, usually getty */
       char *ty_type;   /* terminal type for termcap (3X) */
       int  ty_status;  /* status flags (see below for defines) */
       char *ty_window; /* command to start up window manager */
       char *ty_comment;/* usually the location of the terminal */
       };

#define TTY_ON      0x1  /* enable logins (startup getty) */
#define TTY_SECURE  0x2  /* allow root to login */
#define TTY_LOCAL   0x4  /* line is local direct connect and
                            should ignore modem signals */
#define TTY_SHARED  0x8  /* line is shared - i.e. can be use
                            for both incoming and outgoing
                            connections. */
#define TTY_TRACK   0x10 /* track modem status changes */
#define TTY_TERMIO  0x20 /* open line with termio defaults */

extern struct ttyent *getttyent();
extern struct ttyent *getttynam();
```

A description of the fields follows:

ty_name is the name of the terminal's special file in the directory `/dev`.

ty_getty is the command invoked by `init` to initialize terminal line characteristics. This command is usually `getty`(8), but any arbitrary command can be used. A typical use is to initiate a terminal emulator in a window system.

ty_type is the name of the default terminal type connected to this tty line. This is typically a name from the `termcap`(5) data base. The environment variable 'TERM' is initialized with this name by `login`(1).

ty_status is a mask of bit flags that indicate various actions allowed on this terminal line. The following is a description of each flag.

TTY_ON
Enables logins. For instance, `init`(8) will start the specified `getty` command on this entry.

TTY_SECURE
Allows root to login on this terminal. TTY_ON must also be included for this to work.

TTY_LOCAL
Indicates that the line is to ignore modem signals.

TTY_SHARED
Indicates that the line can be used for both incoming and outgoing connections.

TTY_TERMIO
Indicates that a line is to be opened with default terminal attributes which are compliant with System Five termio defaults. The line discipline will be set to be TERMIODISC.

ty_window
is the quoted string of a command to execute for a window system associated with the line. If none is specified, this will be a null string.

ty_comment
Currently unused.

Restrictions

The information returned is in a static area, so you must copy it to save it. (Static areas are described in "The C Programming Language," *ULTRIX Supplementary Documents*, Vol. II:Programmers.)

Return Value

A null pointer (0) is returned on an end-of-file or error.

Files

`/etc/ttys` The file examined by these routines.

See Also

ttyname(3), ttys(5), init(8)

Name

getwd – get current working directory pathname

Syntax

char *getwd(pathname)
char *pathname;

Description

The `getwd` subroutine copies the absolute pathname of the current working directory to *pathname* and returns a pointer to the result.

Restrictions

The `getwd` subroutine may fail to return to the current directory if an error occurs.

Pathnames can be no longer than MAXPATHLEN as defined in <sys/param.h>.

Return Value

The `getwd` subroutine returns zero and places a message in *pathname* if an error occurs.

Name

hes_init, hes_to_bind, hes_error, hes_resolve - routines for using Hesiod

Syntax

#include <hesiod.h>

hes_init()

char *hes_to_bind(*HesiodName***, *HesiodNameType*)**
char ****HesiodName***, ****HesiodNameType*****;**

hes_error()

har **hes_resolve(*HesiodName***, *HesiodNameType*)**
char ****HesiodName***, ****HesiodNameType*****;**

Description

The `hes_init()` routine opens and reads the Hesiod configuration file, `/etc/hesiod.conf` to extract the left hand side and right hand side of the Hesiod name.

The `hes_to_bind()` routine takes as arguments a `HesiodName` and `HesiodNameType` and returns a fully qualified name to be handed to BIND.

The two most useful routines to the applications programmer are `hes_error()` and `hes_resolve()`. The `hes_error()` routine has no arguments and returns an integer which corresponds to a set of errors which can be found in `hesiod.h` file.

```
#define HES_ER_UNINIT           -1

#define HES_ER_OK                0

#define HES_ER_NOTFOUND          1

#define HES_ER_CONFIG            2

#define HES_ER_NET               3
```

The `hes_resolve()` routine resolves given names via the Hesiod name server. It takes as arguments a name to be resolved, the `HesiodName`, and a type corresponding to the name, the `HesiodNameType`, and returns a pointer to an array of strings which contains all data that matched the query, one match per array slot. The array is null terminated.

If applications require the data to be maintained throughout multiple calls to `hes_resolve()`, the data should be copied since another call to `hes_resolve()` will overwrite any previously-returned data. A null is returned if the data cannot be found.

Examples

The following example shows the use of the Hesiod routines to obtain a Hesiod name from a Hesiod database:

```
#include <hesiod.h>

char *HesiodName, *HesiodNameType;
char **hp;

hp = hes_resolve(HesiodName, HesiodNameType);
if (hp == NULL) {
       error = hes_error();
       switch(error) {
                      .
                      .
                      .
              }
       }
       else
              process(hp);
```

Files

```
/etc/hesiod.conf
/usr/include/hesiod.h
```

See Also

hesiod.conf(5), bindsetup(8)
Guide to the BIND/Hesiod Service

Name

hsearch, hcreate, hdestroy – manage hash search tables

Syntax

```
#include <search.h>

ENTRY *hsearch (item, action)
ENTRY item;
ACTION action;

int hcreate (nel)
unsigned nel;

void hdestroy ( )
```

Description

The `hsearch` subroutine is a hash-table search routine generalized from Knuth (6.4) Algorithm D. It returns a pointer into a hash table indicating the location at which an entry can be found. The *item* is a structure of type ENTRY (defined in the <search.h> header file) containing two pointers: *item.key* points to the comparison key, and *item.data* points to any other data to be associated with that key. (Pointers to types other than character should be cast to pointer-to-character.) The *action* is a member of an enumeration type ACTION indicating the disposition of the entry if it cannot be found in the table. ENTER indicates that the item should be inserted in the table at an appropriate point. FIND indicates that no entry should be made. Unsuccessful resolution is indicated by the return of a NULL pointer.

The `hcreate` subroutine allocates sufficient space for the table, and must be called before `hsearch` is used. The *nel* is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by the algorithm in order to obtain certain mathematically favorable circumstances.

The `hdestroy` subroutine destroys the search table, and may be followed by another call to `hcreate`.

Restrictions

Only one hash search table may be active at any given time.

Diagnostics

The `hsearch` subroutine returns a NULL pointer if either the action is FIND and the item could not be found or the action is ENTER and the table is full.

The `hcreate` subroutine returns zero if it cannot allocate sufficient space for the table.

See Also

bsearch(3), lsearch(3), string(3), tsearch(3)

Name

insque, remque – insert/remove element from a queue

Syntax

```
struct qelem {
struct   qelem *q_forw;
struct   qelem *q_back;
char     q_data[];
};
```

insque(*elem, pred*)
struct qelem **elem, *pred*;

remque(*elem*)
struct qelem **elem*;

Description

The `insque` and `remque` subroutines manipulate queues built from doubly linked lists. Each element in the queue must in the form of ''struct qelem.'' The `insque` subroutine inserts *elem* in a queue immediately after *pred*. The `remque` subroutine removes an entry *elem* from a queue.

Name

isnan – test for NaN

Syntax

#include <math.h>

int isnan (*x*)
double *x*;

Description

The `isnan` function returns 1 if *x* is NaN (the IEEE floating point reserved not-a-number value) and zero otherwise. On VAX, the return value is always zero.

Name

l3tol, ltol3 – convert between 3-byte integers and long integers

Syntax

```
void l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

void ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

The `l3tol` subroutine converts a list of *n* three-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

The `ltol3` performs the reverse conversion from long integers (*lp*) to three-byte integers (*cp*).

These functions are useful for file-system maintenance where the block numbers are three bytes long.

Restrictions

Because of possible differences in byte ordering, the numerical values of the long integers are machine-dependent.

See Also

fs(5)

lockf(3)

Name

lockf – record locking on files

Syntax

#include <unistd.h>

lockf(*fildes*, *function*, *size*)
long *size*;
int *fildes*, *function*;

Description

The `lockf` subroutine allows sections of a file to be locked. These are advisory mode locks. Locking calls from other processes which attempt to lock the locked file section return either an error value or are put to sleep until the resource becomes unlocked. All the locks for a process are removed when the process terminates. For more information about record locking, see `fcntl`(2).

The *fildes* is an open file descriptor. The file descriptor must have O_WRONLY or O_RDWR permission in order to establish lock with this function call.

The *function* is a control value which specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> as follows:

```
#define F_ULOCK 0 /* Unlock a previously locked section */
#define F_LOCK  1 /* Lock a section for exclusive use */
#define F_TLOCK 2 /* Test and lock a section for exclusive use */
#define F_TEST  3 /* Test section for other processes locks */
```

All other values of *function* are reserved for future extensions and result in an error return if not implemented.

F_TEST is used to detect if a lock by another process is present on the specified section. F_LOCK and F_TLOCK both lock a section of a file if the section is available. F_UNLOCK removes locks from a section of the file.

The *size* is the number of contiguous bytes to be locked or unlocked. The resource to be locked or unlocked starts at the current offset in the file and extends forward for a positive size and backward for a negative size. If *size* is zero, the section from the current offset through the largest file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file in order to be locked, as such locks may exist past the end-of-file.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent sections occur, the sections are combined into a single section. If the request requires that a new element be added to the table of active locks and this table is already full, an error is returned, and the new section is not locked.

F_LOCK and F_TLOCK requests differ only by the action taken if the resource is not available. F_LOCK causes the calling process to sleep until the resource is available. F_TLOCK causes the function to return a –1 and set *errno* to [EACCES] error if the section is already locked by another process.

F_ULOCK requests may, in whole or in part, release one or more locked sections controlled by the process. When sections are not fully released, the remaining sections are still locked by the process. Releasing the center section of a locked section requires an additional element in the table of active locks. If this table is full, an [EDEADLK] error is returned and the requested section is not released.

A potential for deadlock occurs if a process controlling a locked resource is put to sleep by accessing another process's locked resource. Thus calls to `lock` or `fcntl` scan for a deadlock prior to sleeping on a locked resource. An error return is made if sleeping on the locked resource would cause a deadlock.

Sleeping on a resource is interrupted with any signal. You can use the `alarm`(3) command to provide a timeout facility in applications which require this facility.

File region locking is supported over NFS, if the NFS locking service has been enabled.

Restrictions

Unexpected results may occur in processes that do buffering in the user address space. The process may later read or write data which is or was locked. The standard I/O package is the most common source of unexpected buffering.

Return Value

Upon successful completion, 0 is returned. Otherwise, a –1 is returned and the global variable *errno* is set to indicate the error.

Diagnostics

The `lockf` subroutine fails if:

[EBADF]	The *fildes* is not a valid open descriptor.
[EACCESS]	The *cmd* is F_TLOCK or F_TEST and the section is already locked by another process. Or, the file is remotely mounted, and the NFS locking service has not been enabled.
[EDEADLK]	The *cmd* is F_LOCK or F_TLOCK and a deadlock would occur. Also the *cmd* is either of the above or F_ULOCK and the number of entries in the lock table would exceed the number allocated on the system.
[EINVAL]	The value given for the *request* argument is invalid.

See Also

close(2), creat(2), fcntl(2), intro(2), open(2), read(2), write(2), lockd(8c)

Name

lsearch, lfind – linear search and update

Syntax

```
#include <search.h>
#include <sys/types.h>

void *lsearch (key, base, nelp, width, compar)
void *key;
void *base;
size_t *nelp;
size_t width;
int (*compar)( );

void *lfind (key, base, nelp, width, compar)
void *key;
void *base;
size_t *nelp;
size_t width;
int (*compar)( );
```

Description

The `lsearch` subroutine is a linear search routine generalized from Knuth (6.1) Algorithm S. It returns a pointer into a table indicating where a datum may be found. If the datum does not occur, it is added at the end of the table. The *key* points to the datum to be sought in the table. The *base* points to the first element in the table. The *nelp* points to an integer containing the current number of elements in the table. The *width* is the size of an element in bytes. The integer is incremented if the datum is added to the table. The *compar* is the name of the comparison function which the user must supply (strcmp, for example). It is called with two arguments that point to the elements being compared. The function must return zero if the elements are equal and non-zero otherwise.

The `lfind` subroutine is the same as lsearch except that if the datum is not found, it is not added to the table. Instead, a NULL pointer is returned.

NOTE

The pointers to the key and the element at the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Restrictions

Undefined results can occur if there is not enough room in the table to add a new item.

Return Value

If the searched for datum is found, both `lsearch` and `lfind` return a pointer to it. Otherwise, `lfind` returns NULL and `lsearch` returns a pointer to the newly added element.

See Also

bsearch(3), hsearch(3), tsearch(3)

Name

malloc, free, realloc, calloc, alloca – memory allocator

Syntax

char *malloc(size)
unsigned size;

free(ptr)
void *ptr;

char *realloc(ptr, size)
void *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

char *alloca(size)
int size;

Description

The `malloc` and `free` subroutines provide a simple general-purpose memory allocation package. The `malloc` subroutine returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to `free` is a pointer to a block previously allocated by `malloc`. This space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by `malloc` is overrun or if some random number is handed to `free`.

The `malloc` subroutine maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls `sbrk` to get more memory from the system when there is no suitable space already free. For further information, see `brk`(2).

The `realloc` subroutine changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

In order to be compatible with older versions, `realloc` also works if *ptr* points to a block freed since the last call of `malloc, realloc,` or `calloc`. Sequences of `free, malloc,` and `realloc` were previously used to attempt storage compaction. This procedure is no longer recommended.

The `calloc` subroutine allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `alloca` subroutine allocates *size* bytes of space associated with the stack frame of the caller. This temporary space is available for reuse when the caller returns. On MIPS machines, calling `alloca`(0) reclaims all available storage. On VAX machines, the space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Restrictions

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

Currently, the allocator is unsuitable for direct use in a large virtual environment where many small blocks are kept, since it keeps all allocated and freed blocks on a circular list. Just before more memory is allocated, all allocated and freed blocks are referenced.

Because the `alloca` subroutine is machine dependent, its use should be avoided.

Diagnostics

The `malloc, realloc,` and `calloc` subroutines return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

Name

malloc, free, realloc, calloc, alloca – memory allocator

Syntax

#include <stdlib.h>

void *malloc(*size***)**
size_t *size***;**

free(*ptr***)**
void ***ptr*;**

void *realloc(*ptr, size***)**
void ***ptr*;**
size_t *size***;**

void *calloc(*nelem, elsize***)**
size_t *nelem, elsize***;**

void *alloca(*size***)**
size_t *size***;**

Description

The `malloc` and `free` subroutines provide a simple general-purpose memory allocation package. The `malloc` subroutine returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to `free` is a pointer to a block previously allocated by `malloc`. This space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by `malloc` is overrun or if some random number is handed to `free`.

The `malloc` subroutine maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls `sbrk` to get more memory from the system when there is no suitable space already free. For further information, see `brk`(2).

The `realloc` subroutine changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

If *ptr* is a null pointer, then `realloc` behaves like `malloc` for the specified *size*. If *size* is zero, then `realloc` frees the space pointed to by *ptr*.

In order to be compatible with older versions, `realloc` also works if *ptr* points to a block freed since the last call of `malloc`, `realloc`, or `calloc`. Sequences of `free`, `malloc`, and `realloc` were previously used to attempt storage compaction. This procedure is no longer recommended.

The `calloc` subroutine allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

The `alloca` subroutine allocates *size* bytes of space in the stack frame of the caller. This temporary space is automatically freed on return.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

Restrictions

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

Currently, the allocator is unsuitable for direct use in a large virtual environment where many small blocks are kept, since it keeps all allocated and freed blocks on a circular list. Just before more memory is allocated, all allocated and freed blocks are referenced.

The `alloca` subroutine is machine dependent.

Diagnostics

The `malloc, realloc,` and `calloc` subroutines return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

The `malloc, realloc, calloc,` and `alloca` subroutines will fail and no additional memory will be allocated if one of the following is true:

[ENOMEM] The limit, as set by `setrlimit`(2), is exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) is exceeded.

[ENOMEM] Insufficient space exists in the swap area to support the expansion.

Name

memccpy, memchr, memcmp, memcpy, memmove, memset – memory operations

Syntax

```
#include <string.h>

void *memccpy (s1, s2, c, n)
void *s1, *s2;
int c;
size_t n;

void *memchr (s, c, n)
void *s;
int c;
size_t n;

int memcmp (s1, s2, n)
void *s1, *s2;
size_t n;

void *memcpy (s1, s2, n)
void *s1, *s2;
size_t n;

void *memset (s, c, n)
void *s;
int c;
size_t n;

void *memmove (s1, s2, n)
void *s1, *s2;
size_t n;
```

Description

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

The `memccpy` subroutine copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied, or after *n* characters have been copied, whichever comes first. It returns a pointer to the character after the copy of *c* in *s1*, or a NULL pointer if *c* was not found in the first *n* characters of *s2*.

The `memchr` subroutine returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a NULL pointer if *c* does not occur.

The `memcmp` subroutine compares its arguments, looking at the first *n* characters only, and returns an integer less than, equal to, or greater than 0, according as *s1* is lexicographically less than, equal to, or greater than *s2*.

The `memcpy` subroutine copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

The memmove subroutine is like memcpy , except that if s1 and s2 specify overlapping areas, memmove works as if an intermediate buffer is used.

The memset subroutine sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

Restrictions

The memcmp subroutine uses native character comparison, which is signed on PDP–11s, unsigned on other machines.

Character movement is performed differently in different implementations of memccpy and memcpy. Thus overlapping moves, using these subroutines, may yield unpredictable results.

Name

mkfifo – make a FIFO special file

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(path, mode)
char *path;
mode_t mode;
```

Description

The `mkfifo` function creates a new FIFO special file whose name is *path*. The file permission bits of the new FIFO are initialized from *mode*, where the value of *mode*, is one (or more) of the file permission bits defined in <sys/stat.h>. The *mode* argument is modified by the process's file creation mask (see `umask`(1)).

The FIFO's owner ID is set to the process's effective user ID. The FIFO's group ID is set to the process's effective group ID.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `mkfifo` function will fail and the FIFO will not be created if:

[EACCES] A component of the path prefix denies search permission.

[EEXIST] The named file exists.

[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire pathname exceeded 1023 characters.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] A component of the path prefix does not exist or the *path* argument points to an empty string.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links were encountered in translating the pathname.

[EIO] An I/O error occurred while making the directory entry.

[ENOSPC] The directory in which the entry for the new FIFO is being placed cannot be extended because there is no space left on the file system.

[ENOSPC] There are no free inodes on the file system on which the node is being created.

[EDQUOT]	The directory in which the entry for the new FIFO is being placed cannot be extended because the user's quota of disk blocks on the file system containing the directory has been exhausted.
[EDQUOT]	The user's quota of inodes on the file system on which the FIFO is being created has been exhausted.
[ESTALE]	The file handle given in the argument is invalid. The file referred to by that file handle no longer exists or has been revoked.
[ETIMEDOUT]	A connect request or remote file operation failed because the connected party did not properly respond after a period of time which is dependent on the communications protocol.

See Also

mknod(1), umask(1)

Name

mktemp – make a unique file name

Syntax

char *mktemp(template)
char *template;

Description

The `mktemp` subroutine replaces *template* by a unique file name, and returns the address of the template. The template should look like a file name with six trailing X's, which will be replaced with the current process ID and a unique letter.

Note: The use of `mktemp` is not recommended for new applications. See `tmpnam`(3) for less error-prone alternatives.

See Also

getpid(2), tmpfile(3), tmpnam(3)

Name

monitor, monstartup, moncontrol – prepare execution profile

Synopsis

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

Description

These functions use the system call `profil`(2) to control program-counter sampling. Using the option **–p** when compiling or linking a program automatically generates calls to these functions. You do need not to call these functions explicitly unless you want more control.

Typically, you would call either `monitor` or `monstartup` to initialize pc-sampling and enable it; call `moncontrol` to disable or reenable it; and call `monitor` at the end of execution to disable sampling and record the samples in a file.

Your initial call to `monitor` enables pc-sampling. The parameters *lowpc* and *highpc* specify the range of addresses to be sampled. The lowest address is that of *lowpc* and the highest is just below *highpc*. The *buffer* parameter is the address of a (user allocated) array of *bufsize* short integers, which holds a record of the samples; for best results, the buffer should not be less than a few times smaller than the range of addresses sampled. The *nfunc* parameter is ignored.

The environment variable PROFDIR determines the name of the output file and whether pc-sampling takes place: if it is not set, the file is named mon.out; if set to the empty string, no pc-sampling occurs; if set to a non-empty string, the file is named string/pid.progname, where pid is the process id of the executing program and progname is the program's name as it appears in argv[0]. The subdirectory string must already exist.

To profile the entire program, use the following:

```
extern eprol(), etext();
. . .
monitor(eprol, etext, buf, bufsize, 0);
```

The routine `eprol` lies just below the user program text, and `etext` lies just above it, as described in `end`(3). (Because the user program does not necessarily start at a low memory address, using a small number in place of `eprol` is dangerous).

The `monstartup` routine is an alternate form of `monitor` that calls `sbrk` (see `brk`(2)) for you to allocate the buffer.

The function `moncontrol` selectively disables and re-enables pc-sampling within a program, allowing you to measure the cost of particular operations. The function `moncontrol(0)` disables pc-sampling, and `moncontrol(1)` reenables it.

To stop execution monitoring and write the results in the output file, use the following:

```
monitor(0);
```

Files

mon.out	default name for output file
libprof1.a	routines for pc-sampling

See Also

cc(1), ld(1), profil(2), brk(2)

Name

monitor, monstartup, moncontrol – prepare execution profile

Syntax

monitor(lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)(), (*highpc)();
short buffer[];

monstartup(lowpc, highpc)
int (*lowpc)(), (*highpc)();

moncontrol(mode)

Description

There are two different forms of monitoring available: An executable program created by:

```
cc -p . . .
```

automatically includes calls for the `prof`(1) monitor and includes an initial call to its start-up routine `monstartup` with default parameters; `monitor` need not be called explicitly except to gain fine control over profil buffer allocation. An executable program created by:

```
cc -pg . . .
```

automatically includes calls for the `gprof`(1) monitor.

The `monstartup` is a high level interface to `profil`(2). The *lowpc* and *highpc* specify the address range that is to be sampled; the lowest address sampled is that of *lowpc* and the highest is just below *highpc*. The `monstartup` subroutine allocates space using `sbrk`(2) and passes it to `monitor` (see below) to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. Only calls of functions compiled with the profiling option **–p** of `cc`(1) are recorded.

To profile the entire program, it is sufficient to use

```
extern etext();
monstartup((int) 2, etext);
```

The *etext* lies just above all the program text, see end(3).

To stop execution monitoring and write the results on the file `mon.out`, use

```
monitor(0);
```

then `prof`(1) can be used to examine the results.

The `moncontrol` subroutine is used to selectively control profiling within a program. This works with either `prof`(1) or `gprof`(1) type profiling. When the program starts, profiling begins. To stop the collection of histogram ticks and call counts use moncontrol(0); to resume the collection of histogram ticks and call counts use moncontrol(1). This allows the cost of particular operations to be measured. Note that an output file will be produced upon program exit regardless of the state of `moncontrol`.

The `monitor` subroutine is a low level interface to `profil`(2). The *lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a (user supplied) array of *bufsize* short integers. At most *nfunc* call counts can be kept. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled. The `monitor` subroutine divides the buffer into space to record the histogram of program counter samples over the range *lowpc* to *highpc*, and space to record call counts of functions compiled with the **–p** option to `cc`(1).

To profile the entire program, it is sufficient to use

```
extern etext();
monitor((int) 2, etext, buf, bufsize, nfunc);
```

Files

mon.out

See Also

cc(1), gprof(1), prof(1), profil(2), sbrk(2)

Name

dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr – data base subroutines

Syntax

```
#include <ndbm.h>

typedef struct {
    char *dptr;
    int dsize;
} datum;

DBM *dbm_open(file, flags, mode)
    char *file;
    int flags, mode;

void dbm_close(db)
    DBM *db;

datum dbm_fetch(db, key)
    DBM *db;
    datum key;

int dbm_store(db, key, content, flags)
    DBM *db;
    datum key, content;
    int flags;

int dbm_delete(db, key)
    DBM *db;
    datum key;

datum dbm_firstkey(db)
    DBM *db;

datum dbm_nextkey(db)
    DBM *db;

int dbm_error(db)
    DBM *db;

int dbm_clearerr(db)
    DBM *db;
```

Description

These functions maintain key/content pairs in a data base. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. This package replaces the earlier dbm(3x) library, which managed only a single database.

The *key*s and *content*s are described by the **datum** typedef. A **datum** specifies a string of **dsize** bytes pointed to by **dptr.** Arbitrary binary data, as well as normal ASCII strings, are allowed. The data base is stored in two files. One file is a directory containing a bit map and has .dir as its suffix. The second file contains all data and has .pag as its suffix.

Before a database can be accessed, it must be opened by **dbm_open**. This will open and/or create the files *file*.dir and *file*.pag depending on the flags parameter (see `open`(2)).

Once open, the data stored under a key is accessed by **dbm_fetch** and data is placed under a key by **dbm_store**. The *flags* field can be either DBM_INSERT or DBM_REPLACE. DBM_INSERT will only insert new entries into the database and will not change an existing entry with the same key. DBM_REPLACE will replace an existing entry if it has the same key. A key (and its associated contents) is deleted by **dbm_delete**. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of **dbm_firstkey** and **dbm_nextkey**. **dbm_firstkey** will return the first key in the database. **dbm_nextkey** will return the next key in the database. This code will traverse the data base:

> **for** (*key* = **dbm_firstkey**(*db*); *key*.**dptr** != NULL; *key* = **dbm_nextkey**(*db*))

dbm_error returns non-zero when an error has occurred reading or writing the database. **dbm_clearerr** resets the error condition on the named database.

Diagnostics

All functions that return an **int** indicate errors with negative values. A zero return indicates ok. Routines that return a **datum** indicate errors with a null (0) **dptr.** If **dbm_store** called with a *flags* value of DBM_INSERT finds an existing entry with the same key it returns 1.

Restrictions

The '.pag' file will contain holes so that its apparent size is about four times its actual content. Older systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (`cp, cat, tp, tar, ar`) without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 4096 bytes). Moreover all key/content pairs that hash together must fit on a single block. **dbm_store** will return an error in the event that a disk block fills with inseparable data.

dbm_delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by **dbm_firstkey** and **dbm_nextkey** depends on a hashing function, not on anything interesting.

See Also

`dbm`(3X)

Name

nice – set program priority

Syntax

nice(incr)

Description

The scheduling priority of the process is augmented by *incr*. Positive priorities get less service than normal. Priority 10 is recommended to users who wish to execute long-running programs without flack from the administration.

Negative increments are ignored except on behalf of the super-user. The priority is limited to the range –20 (most urgent) to 20 (least).

The priority of a process is passed to a child process by `fork`(2). For a privileged process to return to normal priority from an unknown state, `nice` should be called successively with arguments –40 (goes to priority –20 because of truncation), 20 (to get to 0), then 0 (to maintain compatibility with previous versions of this call).

Environment

When your program is compiled using the System V environment, upon success, `nice` returns –20.

See Also

nice(1), fork(2), setpriority(2), renice(8)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See `a.out`(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file `/vmunix`. In this way programs can obtain system addresses that are up to date.

Diagnostics

If the file cannot be found or if it is not a valid namelist –1 is returned; otherwise, the number of unfound namelist entries is returned.

The type entry is set to 0 if the symbol is not found.

See Also

a.out(5)

Name

nlist – get entries from name list

Syntax

#include <nlist.h>

nlist(filename, nl)
char *filename;
struct nlist nl[];

Description

The `nlist` subroutine examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See `a.out`(5) for the structure declaration.

This subroutine is useful for examining the system name list kept in the file `/vmunix`. In this way programs can obtain system addresses that are up to date.

Diagnostics

All type entries are set to 0 if the file cannot be found or if it is not a valid name list.

See Also

a.out(5)

pathconf(3)

Name

pathconf, fpathconf – get configurable pathname variables (POSIX)

Syntax

#include <unistd.h>

long pathconf(*path, name***)**
char ***path*;**
int *name***;**

long fpathconf(*fildes, name***)**
int *fildes, name***;**

Description

The `pathconf`(3) and `fpathconf`(3) functions provide a method for the application to determine the current value of a configurable limit or option that is associated with a file or directory.

For `pathconf`(3), the *path* argument points to the pathname of a file or directory. For `fpathconf`(3), the *fildes* argument is an open file descriptor.

The *name* argument represents the variable to be queried relative to that file or directory. The following table lists the variables which may be queried and the corresponding value for the *name* argument. The values for the *name* argument are defined in the <unistd.h> header file.

Variable	name Value
LINK_MAX	_PC_LINK_MAX
MAX_CANON	_PC_MAX_CANON
MAX_INPUT	_PC_MAX_INPUT
NAME_MAX	_PC_NAME_MAX
PATH_MAX	_PC_PATH_MAX
PIPE_BUF	_PC_PIPE_BUF
_POSIX_CHOWN_RESTRICTED	_PC_CHOWN_RESTRICTED
_POSIX_NO_TRUNC	_PC_NO_TRUNC
_POSIX_VDISABLE	_PC_VDISABLE

Return Value

Upon successful completion, the `pathconf`(3) and `fpathconf`(3) functions return the current variable value for the file or directory.

If *name* is an invalid value, `pathconf`(3) and `fpathconf`(3) return –1 and *errno* is set to indicate the reason. If the variable corresponding to *name* is not defined on the system, `pathconf`(3) and `fpathconf`(3) return –1 without changing the value of *errno*.

Diagnostics

The `pathconf`(3) and `fpathconf`(3) functions fail if the following occurs:

[EINVAL] The value of the *name* argument is invalid.

See Also

<unistd.h>

pause(3)

Name

pause – stop until signal

Syntax

pause()

Description

The `pause` subroutine never returns normally. It is used to give up control while waiting for a signal from `kill`(2) or an interval timer, see `setitimer`(2). Upon termination of a signal handler started during a `pause`, the `pause` call will return.

Diagnostics

The `pause` subroutine always returns:

[EINTR] The call was interrupted, that is, always returns –1.

See Also

kill(2), select(2), sigpause(2)

Name

perror, strerror, sys_errlist, sys_nerr – system error messages

Syntax

perror(*s***)**
char * *s;*

int sys_nerr;
char *sys_errlist[];

#include <string.h>

char *strerror(*err***)**
int *err;*

Description

The `perror` subroutine produces a short error message on the standard error file describing the last error encountered during a call to the system from a C program. First the argument string *s* , if it is not a null pointer, is printed followed by a colon and a space; then the message and a new line are printed. Most usefully, the argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno* which is set when errors occur but not cleared when nonerroneous calls are made. For further information, see `intro`(2).

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new line. The *sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table. The `strerror` function will also return a pointer to the message text for a given error number.

See Also

intro(2), `errno`(2), psignal(3)

pfopen(3)

Name

pfopen – open a packet filter file

Syntax

pfopen(ifname, flags)
char *ifname;
int flags;

Description

The packet filter (see `packetfilter`(4)) provides raw access to Ethernets and similar network data link layers. The routine `pfopen` is used to open a packet filter file descriptor. The routine hides various details about the way packet filter files are opened and named.

The *ifname* argument is a pointer to a null-terminated string containing the name of the interface for which the application is opening the packet filter. This name may be the name of an actual interface on the system (for example, ''de0'', ''qe2'') or it may be a pseudo-interface name of the form ''pf*n*'', used to specify the *n*th interface attached to the system. For example, ''pf0'' specifies the first such interface. If *ifname* is NULL, the default interface (''pf0'') is used.

The *flags* argument has the same meaning as the corresponding argument to the `open`(2) system call.

The file descriptor returned by `pfopen` is otherwise identical to one returned by `open`(2).

Diagnostics

The `pfopen` routine returns a negative integer if the file could not be opened. This may be because of resource limitations, or because the specified interface does not exist.

If there are a lot of packet filter applications in use, the `pfopen` routine might take a while.

See Also

open(2), packetfilter(4)
The Packet Filter: An Efficient Mechanism for User Level Network Code

Name

popen, pclose – initiate I/O to/from a process

Syntax

#include <stdio.h>

FILE *popen(command, type)
char *command, *type;

pclose(stream)
FILE *stream;

Description

The arguments to `popen` are pointers to null-terminated strings containing respectively a shell command line and an I/O mode, either "r" for reading or "w" for writing. It creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by `popen` should be closed by `pclose`, which waits for the associated process to terminate and returns the exit status of the command.

Because open files are shared, a type "r" command may be used as an input filter, and a type "w" as an output filter.

Environment

Differs from the System V definition in that ENFILE is not a possible error condition.

Diagnostics

The `popen` routine returns a null pointer if files or processes cannot be created, or the shell cannot be accessed.

The `pclose` routine returns –1 if *stream* is not associated with a 'popened' command.

Restrictions

Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing, for instance, with `fflush`. For further information, see `fclose`(3).

The `popen` routine always calls `sh`, and never calls `csh`.

See Also

sh(1), pipe(2), wait(2), system(3), fclose(3s), fopen(3s)

psignal(3)

Name

psignal, sys_siglist – system signal messages

Syntax

psignal(sig, s)
unsigned sig;
char *s;

char *sys_siglist[];

Description

The `psignal` subroutine produces a short message on the standard error file describing the indicated signal. First the argument string *s* is printed, then a colon, then the name of the signal and a new-line. Most usefully, the argument string is the name of the program which incurred the signal. The signal number should be from among those found in <signal.h>.

To simplify variant formatting of signal names, the vector of message strings `sys_siglist` is provided. The signal number can be used as an index in this table to get the signal name without the newline. The define NSIG defined in <signal.h> is the number of messages.

See Also

sigvec(2), perror(3)

Name

putenv – change or add value to environment

Syntax

int putenv (string)
char *string;

Description

The *string* points to a string of the form *"name=value."* The `putenv` subroutine makes the value of the environment variable *name* equal to *value* by altering an existing variable or creating a new one. In either case, the string pointed to by *string* becomes part of the environment, so altering the string will change the environment. The space used by *string* is no longer used once a new string-defining *name* is passed to `putenv`.

Diagnostics

The `putenv` subroutine returns nonzero if it was unable to obtain enough space via `malloc` for an expanded environment, otherwise zero.

Warnings

The `putenv` subroutine manipulates the environment pointed to by `environ`, and can be used in conjunction with `getenv`. However, *envp* (the third argument to *main*) is not changed.
This routine uses `malloc`(3) to enlarge the environment.
After `putenv` is called, environmental variables are not in alphabetical order.
A potential error is to call `putenv` with an automatic variable as the argument, then exit the calling function while *string* is still part of the environment.

See Also

execve(2), getenv(3), malloc(3), environ(7)

Name

putpwent – write password file entry

Syntax

#include <pwd.h>

int putpwent (*p, f*)
struct passwd **p*;
FILE **f*;

Description

The `putpwent` subroutine is the inverse of `getpwent`(3). Given a pointer to a *passwd* structure created by `getpwent` (or `getpwuid` or `getpwnam`), `putpwent` writes a line on the stream *f* which matches the format of `/etc/passwd`.

Diagnostics

The `putpwent` subroutine returns non-zero if an error was detected during its operation, otherwise zero.

Caution

The `putpwent` routine uses <stdio.h>, which causes it to increase the size of programs, not otherwise using standard I/O, more than might be expected.

Name

qsort – quicker sort

Syntax

#include <stdlib.h>

void qsort(*base, nel, width, compar*)
void ***base*;
size_t *nel, width*;
int (***compar*)();

Description

The `qsort` subroutine is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine to be called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according as the first argument is to be considered less than, equal to, or greater than the second.

See Also

sort(1)

Name

rand, srand – random number generator

Syntax

#include <stdlib.h>

srand(*seed***)**
unsigned *seed***;**

rand()

Description

The newer `random`(3) should be used in new applications. The `rand` subroutine remains for compatibility.

The `rand` subroutine uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{31}-1$.

The generator is reinitialized by calling `srand` with 1 as argument. It can be set to a random starting point by calling `srand` with whatever you like as argument.

Environment

For the System V environment, the `rand` subroutine returns numbers in the range from 0 to $2^{15}-1$.

See Also

random(3)

Name

random, srandom, initstate, setstate – better random number generator; routines for changing generators

Syntax

long random()

srandom(seed)
int seed;

char *initstate(seed, state, n)
unsigned seed;
char *state;
int n;

char *setstate(state)
char *state;

Description

The `random` subroutine uses a non-linear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range from 0 to (2**31)-1. The period of this random number generator is very large, approximately 16*((2**31)-1).

The `random/srandom` subroutines have (almost) the same calling sequence and initialization properties as `rand/srand`. The difference is that rand(3) produces a much less random sequence – in fact, the low dozen bits generated by rand go through a cyclic pattern. All the bits generated by `random` are usable. For example, ''random()&01'' will produce a random binary value.

Unlike `srand`, `srandom` does not return the old seed; the reason for this is that the amount of state information used is much more than a single word. (Two other routines are provided to deal with restarting/changing random number generators.) Like `rand`(3), however, `random` will by default produce a sequence of numbers that can be duplicated by calling `srandom` with *1* as the seed.

The `initstate` routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by `initstate` to decide how sophisticated a random number generator it should use – the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error). The seed for the initialization (which specifies a starting point for the random number sequence, and provides for restarting at the same point) is also an argument. `Initstate` returns a pointer to the previous state information array.

Once a state has been initialized, the `setstate` routine provides for rapid switching between states. The `setstate` subroutine returns a pointer to the previous state array; its argument state array is used for further random number generation until the next call to `initstate` or `setstate`.

Once a state array has been initialized, it may be restarted at a different point either by calling `initstate` (with the desired seed, the state array, and its size) or by calling both `setstate` (with the state array) and `srandom` (with the desired seed).

The advantage of calling both `setstate` and `srandom` is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

Diagnostics

If `initstate` is called with less than 8 bytes of state information, or if `setstate` detects that the state information has been garbled, error messages are printed on the standard error output.

See Also

rand(3)

Name

re_comp, re_exec – regular expression handler

Syntax

char *re_comp(s)
char *s;

re_exec(s)
char *s;

Description

The `re_comp` subroutine compiles a string into an internal form suitable for pattern matching. The `re_exec` subroutine checks the argument string against the last string passed to `re_comp`.

The `re_comp` subroutine returns 0 if the string *s* was compiled successfully; otherwise a string containing an error message is returned. If `re_comp` is passed 0 or a null string, it returns without changing the currently compiled regular expression.

The `re_exec` subroutine returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression was invalid (indicating an internal error).

The strings passed to both `re_comp` and `re_exec` may have trailing or embedded newline characters; they are terminated by nulls. The regular expressions recognized are described in the manual entry for `ed`(1), given the above difference.

Diagnostics

The `re_exec` subroutine returns –1 for an internal error.

The `re_comp` subroutine returns one of the following strings if an error occurs:

No previous regular expression
Regular expression too long
unmatched \(
missing]
too many \(\) pairs
unmatched \)

See Also

ed(1), ex(1), egrep(1), fgrep(1), grep(1)

remove(3)

Name

remove – removes files

Syntax

remove (*path*)
char *path;

Arguments

path	Provides the specification for a file or directory.

Description

The `remove` library function removes a file. If the *path* does not name a directory then *remove(path)* is equivalent to *unlink(path)*. If the *path* does name a directory then *remove(path)* is equivalent to *rmdir(path)*.

Return Value

A 0 is returned if the remove succeeds; otherwise a –1 is returned and an error code is stored in the global location *errno*.

See Also

errno(2), rmdir(2), unlink(2)

Name

res_mkquery, res_send, res_init, dn_comp, dn_expand – resolver routines

Syntax

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

res_mkquery(op, dname, class, type, data, datalen, newrr, buf, buflen)
int op;
char *dname;
int class, type;
char *data;
int datalen;
struct rrec *newrr;
char *buf;
int buflen;

res_send(msg, msglen, answer, anslen)
char *msg;
int msglen;
char *answer;
int anslen;

res_init()

dn_comp(exp_dn, comp_dn, length, dnptrs, lastdnptr)
char *exp_dn, *comp_dn;
int length;
char **dnptrs, **lastdnptr;

dn_expand(msg, eomorig, comp_dn, exp_dn, length)
char *msg, *eomorig, *comp_dn, exp_dn;
int length;
```

Description

The resolver routines are used for making, sending, and interpreting packets to BIND servers. Global information that is used by the resolver routines is kept in the variable `_res`. Most of the values have reasonable defaults and you need not be concerned with them. The options are a simple bit mask and are or'ed in to enable. The options stored in `_res.options` are defined in `/usr/include/resolv.h` and are as follows:

RES_INIT	True if the initial name server address and default domain name are initialized, for example if `res_init` has been called.
RES_DEBUG	Print debugging messages.
RES_AAONLY	Accept authoritative answers only.
RES_USEVC	Use TCP connections for queries instead of UDP.
RES_STAYOPEN	This is used with `RES_USEVC` to keep the TCP connection

open between queries. This is useful only in programs that regularly do many queries. You should normally use UDP.

RES_RECURSE Set the recursion desired bit in queries. This is the default. The `res_send` routine does not do iterative queries and expects the BIND server to handle recursion.

RES_DEFNAMES Append the default domain name to single label queries. This is the default.

The following lists the routines found in `/usr/lib/libc.a`

res_init This routine reads the initialization file to get the default domain name and the Internet address of the initial hosts running the name server. If this line does not exist, the host running the resolver is tried.

res_mkquery This routine makes a standard query message and places it in *buf*. The *res_mkquery* routine returns the size of the query or –1 if the query is larger than *buflen*.

op The opcode is usually `QUERY`, but can be any of the query types defined in *nameser.h*.

Dname
This variable is the domain name. If *dname* consists of a single label and the `RES_DEFNAMES` flag is enabled, which is the default, *dname* is appended with the current domain name. The current domain name is defined in a system file, but you can override it by using the environment variable `LOCALDOMAIN`.

res_send This routine sends a query to the BIND servers and returns an answer. It calls the `res_init` routine. If `RES_INIT` is not set, `res_send` sends the query to the local name server, and handle timeouts and retries. The length of the message is returned or –1 if there were errors.

dn_comp This routine compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or –1 if there were errors. The *length* is the size of the array pointed to by *comp_dn*.

dnptrs
This variable is a list of pointers to previously compressed names in the current message. The first pointer points to the beginning of the message and the list ends with `NULL`.

lastdnptr
This is a pointer to the end of the array pointed to by *dnptrs*. A side effect is to update the list of pointers for labels inserted into the message by *dn_comp* as the name is compressed. If *dnptr* is NULL, the names are not compressed. If *lastdnptr* is NULL, the list is not updated.

dn_expand This routine expands the compressed domain name `comp_dn` to a full BIND domain name. Expanded names are converted to upper case.

msg This variable is a pointer to the beginning of the message.

exp_dn
This variable is a pointer to a buffer of size *length* for the result. The size of the compressed name is returned or -1 if there was an error.

Files

```
/etc/resolv.conf
/usr/include/resolv.h
/usr/include/arpa/nameser.h
```

See Also

named(8), resolv.conf(5)
Guide to the BIND/Hesiod Service

scandir(3)

Name

scandir – scan a directory

Syntax

```
#include <sys/types.h>
#include <sys/dir.h>

scandir(dirname, namelist, select, compar)
char *dirname;
struct direct *(*namelist[]);
int (*select)();
int (*compar)();

alphasort(d1, d2)
struct direct **d1, **d2;
```

Description

The `scandir` subroutine reads the directory *dirname* and builds an array of pointers to directory entries using `malloc`(3). It returns the number of entries in the array and a pointer to the array through *namelist*.

The *select* parameter is a pointer to a user supplied subroutine which is called by `scandir` to select which entries are to be included in the array. The select routine is passed a pointer to a directory entry and should return a non-zero value if the directory entry is to be included in the array. If *select* is null, then all the directory entries will be included.

The *compar* parameter is a pointer to a user supplied subroutine which is passed to `qsort`(3) to sort the completed array. If this pointer is null, the array is not sorted. The `alphasort` is a routine which can be used for the *compar* parameter to sort the array alphabetically.

The memory allocated for the array can be deallocated with *free* by freeing each pointer in the array and the array itself. For further information, see `malloc`(3).

Diagnostics

Returns –1 if the directory cannot be opened for reading or if `malloc`(3) cannot allocate enough memory to hold all the data structures.

See Also

directory(3), malloc(3), qsort(3), dir(5)

Name

setjmp, longjmp – non-local goto

Syntax

#include <setjmp.h>

int setjmp (env)
jmp_buf env;

void longjmp (env, val)
jmp_buf env;
int val;

Description

The `setjmp` and `longjmp` functions help deal with errors and interrupts encountered in a low-level subroutine of a program.

The `setjmp` function saves its stack environment in *env* (whose type, *jmp_buf*, is defined in the *<setjmp.h>* header file) for later use by `longjmp`. It returns the value 0.

The `longjmp` function restores the environment saved by the last call of `setjmp` with the corresponding *env* argument. After `longjmp` finishes, program execution continues as if the corresponding call of `setjmp` (which must not itself have returned in the interim) had just returned the value *val*. The `longjmp` function cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a second argument of 0, `setjmp` returns 1. At the time of the second return from `setjmp`, all accessible data have values as of the time `longjmp` is called. However, global variables have the expected values. For example, those as of the time of the `longjmp`(see

Examples

```
#include <setjmp.h>

jmp_buf env;
int i = 0;
main ()
{
        void exit();

        if(setjmp(env) != 0) {
                (void) printf("value of i on 2nd return from setjmp: %d0, i);
                exit(0);
        }
        (void) printf("value of i on 1st return from setjmp: %d0, i);
        i = 1;
        g();
        /*NOTREACHED*/
}
```

```
g()
{
        longjmp(env, 1);
        /*NOTREACHED*/
}
```

If the a.out resulting from this C language code is run, the output is as follows:

```
value of i on 1st return from setjmp:0

value of i on 2nd return from setjmp:1
```

NOTE

Unexpected behavior occurs if `longjmp` is called without a previous call to `setjmp`, or when the last such call was in a function which has since returned.

Restrictions

The values of the registers on the second return from `setjmp` are register values at the time of the first call to `setjmp`, not those of the `longjmp`. Thus, variables in a given function can produce unexpected results in the presence of `setjmp`, depending on whether they are register or stack variables.

See Also

signal(2).

Name

setjmp, longjmp – nonlocal goto

Syntax

```
#include <setjmp.h>

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;

_setjmp(env)
jmp_buf env;

_longjmp(env, val)
jmp_buf env;
```

Description

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The `setjmp` subroutine saves its stack environment in *env* for later use by `longjmp`. It returns value 0.

The `longjmp` subroutine restores the environment saved by the last call of `setjmp`. It then returns in such a way that execution continues as if the call of `setjmp` had just returned the value *val* to the function that invoked `setjmp`, which must not itself have returned in the interim. However, `longjmp` cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a *val* of 0, `setjmp` will return 1. All accessible data have values as of the time `longjmp` was called.

The `setjmp` and `longjmp` subroutines save and restore the signal mask sigsetmask(2), while `_setjmp` and `_longjmp` manipulate only the C stack and registers.

Restrictions

The `setjmp` subroutine does not save current notion of whether the process is executing on the signal stack. The result is that a longjmp to some place on the signal stack leaves the signal stack state incorrect.

See Also

sigstack(2), sigvec(2), signal(3)

Name

setlocale – set localization for internationalized program

Syntax

#include <locale.h>

char *setlocale (*category*, *locale*)
int category;
char *locale;

Description

The `setlocale` function changes or queries the run-time environment of the program. The function can affect the settings of language, territory, and codeset in the program's environment.

In the *category* argument, you specify what part of the run-time environment you want to affect. Possible values for *category* are shown in the following table:

category	Effect of Specifying the Value	Environment Variable Affected
LC_ALL	Sets or queries entire environment	LANG
LC_COLLATE	Changes or queries collation sequences	LC_COLLATE
LC_CTYPE	Changes or queries character classification	LC_CTYPE
LC_NUMERIC	Changes or queries number format information	LC_NUMERIC
LC_TIME	Changes or queries time conversion parameters	LC_TIME
LC_MONETARY	Changes or queries monetary information	LC_MONETARY

You change only one part of the program's locale in a single call to `setlocale`, unless you use the category `LC_ALL`.

The *locale* argument is a pointer to a character string containing the required setting of *category* in the following format:

```
language[_territory[.codeset]][@modifier]
```

You use *language* to specify the native language you want in the program environment. You can specify what dialect of the native language you want in *_territory*, and the codeset to be used in *codeset*. For example, the following string specifies the French native language, as spoken in France (as opposed to Switzerland), and the Digital Multinational Character Set:

```
LANG = FRE_FR.MCS
```

You use *@modifier* to select a specific instance of an environment setting within a single category. For example, you could use *@modifier* to select dictionary sorting of data, as opposed to telephone directory sorting. You can use *@modifier* for all categories, except `LC_ALL`.

The following preset values of *locale* are defined for all the settings of *category*:

"C" Specifies setting the locale to the minimum C language environment, as specified by the ANSI standard for the C language. (Draft ANSI X3.159)

"" Specifies using the environment variable corresponding to *category* to set the locale. If the appropriate environment variable is not set, the `LANG` environment variable is used. If `LANG` is not set, `setlocale` returns an error.

NULL Queries the current international environment and returns current locale setting. You can use the string `setlocale` returns only as input to a subsequent `setlocale` call; in particular, the string cannot be printed for category `LC_ALL`. The string `setlocale` returns is a pointer to static data area that might be written over.

International Environment

INTLINFO The INTLINFO environment variable specifies the directory to search for language databases. The default is to search the `/usr/lib/intln` directory.

Examples

The following calls to the `setlocale` function set the environment to the French language and then modify the collating sequence to German dictionary collation:

```
setlocale (LC_ALL, "FRE_FR.MCS");
setlocale (LC_COLLATE, "GER_DE.MCS@dict");
```

You can use the `setlocale` function to bind the specific language requirements of a user to the program as follows:

```
status = setlocale (LC_ALL, "");
```

For this example to work properly, the user of the international program sets the LANG variable before running the program. Once LANG is set and the program runs, this call causes `setlocale` to use the definition of LANG to set the current locale. You should test the value of `status` after the call completes to be sure no errors occur.

Return Values

If you pass valid setting for *category* and *locale*, other than NULL, `setlocale` changes the current locale and returns the string associated with that locale.

If *locale* is NULL, `setlocale` returns the string associated with *category* for the current *locale*. The current *locale* is unchanged. The string `setlocale` returns may not be in a printable format.

If either the *category* or *locale* argument is invalid, `setlocale` returns `NULL`. The `setlocale` function does not modify the locale if any part of the call is invalid.

The `setlocale` function stores its return values in a data area that may be written over. You should move the return value to another location if you want to use it in your program.

See Also

ic(1int), nl_langinfo(3int), printf(3int), environ(5int), lang(5int)
Guide to Developing International Software

Name

setpgid – set process group (POSIX)

Syntax

```
#include <sys/types.h>
int
setpgid(pid, pgrp)
pid_t pid, pgrp;
```

Description

The `setpgid` function is used to either join an existing process group or create a new process group within the session of the calling process (see `setsid`(2)). Upon successful completion, the process group ID of the process that has a process ID which matches *pid* is set to *pgrp*. If *pid* is zero, then the call applies to the current process. In addition, if *pgrp* is zero, the process ID of the indicated process is used.

This function is available only in the POSIX environment.

Return Value

The `setpgid` function returns 0 when the operation is successful. If the request fails, –1 is returned and the global variable `errno` indicates the reason.

Diagnostics

The `setpgid` function fails and the process group is not altered if one of the following occurs:

[EACCES]	The value of the *pid* argument matches the process ID of a child process of the calling process and the child process has successfully executed an `exec` function.
[EINVAL]	The value of the *pgrp* argument is less than zero or is not a supported value.
[EPERM]	The process indicated by the *pid* argument is a session leader. The value of the *pid* argument matches the process ID of a child process of the calling process and the child process is not in the same session as the calling process. The value of the *pgrp* argument does not match the process ID of the process indicated by the *pid* argument and there is no process with a process group ID that matches the value of the *pgrp* argument in the same session as the calling process.
[ESRCH]	The value of the *pid* argument does not match the process ID of the calling process of a child process of the calling process.

See Also

getpgrp(2), setsid(2)

Name

setuid, seteuid, setruid, setgid, setegid, setrgid – set user and group ID

Syntax

#include <*sys/types.h*>
#include <*unistd.h*>

setuid(*uid*)
uid_t *uid*;
seteuid(*euid*)
uid_t *euid*;
setruid(*ruid*)
uid_t *ruid*;

setgid(*gid*)
gid_t *gid*;
setegid(*egid*)
gid_t *egid*;
setrgid(*rgid*)
gid_t *rgid*;

Description

The `setuid` subroutine sets both the real and effective user ID of the current process to the ID specified. Likewise, the `setgid` subroutine sets the real and effective group ID of the current process to the ID specified.

The `seteuid` subroutine sets the effective user ID of the current process, while the `setegid` subroutine sets the effective group ID of the current process.

The `setruid` subroutine sets the real user ID of the current process, while the `setrgid` subroutine sets the real group ID of the current process.

These calls are only permitted to the super-user or if the argument is the real or effective ID.

Environment

POSIX
SYSTEM_FIVE
When your program is compiled in POSIX or System V mode the following semantics apply when using the `setuid` or `setgid` functions:

If the process is the super-user the real, effective, and saved set (as described in `execve`(2)) user/group ID are set to *uid*.

If the process is not the super-user, but *uid* is equal to the real or the saved set user/group ID, the effective user/group ID is set to *uid*. The real and saved set user/group ID remain unchanged.

POSIX
In POSIX mode, the `setuid` function returns a value of type uid_t. The `setgid` function returns a value of type gid_t.

Return Values

Zero is returned if the user ID or group ID is set; –1 is returned otherwise.

See Also

setreuid(2), setregid(2), getuid(2), getgid(2)

Name

sigaction – software signal facilities (POSIX)

Syntax

```
#include <signal.h>

struct sigaction {
        void     (*sa_handler)();
        sigset_t sa_mask;
        int      sa_flags;
};

int sigaction(sig, vec, ovec)
int sig;
struct sigaction *vec, *ovec;
```

Description

The siagaction call is the POSIX equivalent to the `sigvec`(2) system call. This call behaves as described on the `sigvec`(2) reference page with the following modifications:

- The signal mask is manipulated using the `sigsetops`(3) functions.
- A process can suppress the generation of the SIGCHLD when a child stops by setting the SA_NOCLDSTOP bit in *sa_flags*.
- The SV_INTERRUPT flag is always set by the system when using `sigaction`(3) in POSIX mode. The flag is set so that interrupted system calls will fail with the EINTR error instead of getting restarted.

Return Value

A 0 return value indicated that the call succeeded. A –1 return value indicates an error occurred and *errno* is set to indicated the reason.

Diagnostics

The `sigaction` system call fails and a new signal handler is not installed if one of the following occurs:

[EFAULT]	Either *vec* or *ovec* points to memory which is not a valid part of the process address space.
[EINVAL]	*Sig* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

See Also

sigvec(2), sigsetops(3), sigprocmask(3), sigsuspend(3), sigpending(2), setjmp(3), siginterrupt(3), tty(4)

siginterrupt(3)

Name

siginterrupt – allow signals to interrupt system calls

Syntax

siginterrupt(*sig, flag*)
int *sig, flag*;

Description

The `siginterrupt` system call is used to change the system call restart behavior when a system call is interrupted by the specified signal. If the flag is false (0), then system calls will be restarted if they are interrupted by the specified signal and no data has been transferred yet. System call restart is the default behavior on 4.2 BSD.

If the flag is true (1), then restarting of system calls is disabled. If a system call is interrupted by the specified signal and no data has been transferred, the system call will return -1 with errno set to EINTR. Interrupted system calls that have started transferring data will return the amount of data actually transferred. System call interrupt is the signal behavior found on 4.1 BSD and AT&T System V systems.

Note that the new signal handling semantics are not altered in any other way. Most notably, signal handlers always remain installed until explicitly changed by a subsequent `sigvec`(2) call, and the signal mask operates as documented in `sigvec`(2.) Programs may switch between restartable and interruptible system call operation as often as desired in the execution of a program.

Issuing a `siginterrupt` call during the execution of a signal handler will cause the new action to take place on the next signal to be caught.

Environment

This library routine uses an extension of the `sigvec`(2) system call that is not available in ULTRIX 2.0 or earlier versions. Hence it should not be used if backward compatibility is needed.

Return Value

A 0 value indicates that the call succeeded. A -1 value indicates that an invalid signal number has been supplied.

See Also

sigvec(2), sigblock(2), sigpause(2), sigsetmask(2)

Name

signal – simplified software signal facilities

Syntax

#include <signal.h>

(*signal(sig, func))()
void (*func)();

Description

The `signal` subroutine is a simplified interface to the more general sigvec(2) facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background. For further information, see `tty`(4). Signals are optionally generated when a process resumes after being stopped, when the status of child process changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the `signal` call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file < signal.h >:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16•	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19•	Continue after stop
SIGCHLD	20•	Child status has changed
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	Cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	File size limit exceeded (see setrlimit(2))

SIGVTALRM	26	Virtual time alarm (see setitimer(2))
SIGPROF	27	Profiling timer alarm (see setitimer(2))
SIGWINCH	28•	Window size change
SIGUSR1	30	User defined signal
SIGUSR2	31	User defined signal
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or +. Signals marked with • are discarded if the action is SIG_DFL; signals marked with + cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read* or `write`(2) on a slow device (such as a terminal; but not a file) and during a `wait`(2).

The value of `signal` is the previous (or initial) value of *func* for the particular signal.

After a `fork`(2) or `vfork`(2) the child inherits all signals. The `execve`(2) system call resets all caught signals to the default action; ignored signals remain ignored.

Environment

When your program is compiled using the System V environment the handler function does NOT remain installed after the signal has been delivered.

Also, when a signal which is to be caught occurs during a read, write, or ioctl to a slow device (like a terminal, but not a file); or during a pause; or wait that does not return immediately, the signal handler function is executed, and then the interrupted system call may return a -1 to the calling process with errno set to EINTR.

Notes

The handler routine can be declared as follows:

```
handler(sig, code, scp)
int sig, code;
struct sigcontext *scp;
```

Here *sig* is the signal number. The MIPS hardware exceptions are mapped to specific signals as defined by the table below. The parameter *code* is either a constant as given below or zero. The parameter *scp* is a pointer to the *sigcontext* structure (defined in <*signal.h*>), that is the context at the time of the signal and is used to restore the context if the signal handler returns.

The following defines the mapping of MIPS hardware exceptions to signals and codes. All of these symbols are defined in either *<signal.h>* or *<mips/cpu.h>*:

Hardware exception	Signal	Code
Integer overflow	SIGFPE	EXC_OV
Segmentation violation	SIGSEGV	SEXC_SEGV
Illegal Instruction	SIGILL	EXC_II
Coprocessor Unusable	SIGILL	SEXC_CPU
Data Bus Error	SIGBUS	EXC_DBE
Instruction Bus Error	SIGBUS	EXC_IBE
Read Address Error	SIGBUS	EXC_RADE
Write Address Error	SIGBUS	EXC_WADE
User Breakpoint (used by debuggers)	SIGTRAP	BRK_USERBP
Kernel Breakpoint (used by prom)	SIGTRAP	BRK_KERNELBP
Taken Branch Delay Emulation	SIGTRAP	BRK_BD_TAKEN
Not Taken Branch Delay Emulation	SIGTRAP	BRK_BD_NOTTAKEN
User Single Step (used by debuggers)	SIGTRAP	BRK_SSTEPBP
Overflow Check	SIGTRAP	BRK_OVERFLOW
Divide by Zero Check	SIGTRAP	BRK_DIVZERO
Range Error Check	SIGTRAP	BRK_RANGE

When a signal handler is reached, the program counter in the signal context structure (*sc_pc*) points at the instruction that caused the exception as modified by the *branch delay* bit in the *cause* register. The *cause* register at the time of the exception is also saved in the sigcontext structure (*sc_cause*). If the instruction that caused the exception is at a valid user address it can be retrieved with the following code sequence:

```
    if(scp->sc_cause & CAUSE_BD){
        branch_instruction = *(unsigned long *)(scp-
>sc_pc);
        exception_instruction = *(unsigned long *)(scp-
>sc_pc + 4);
    }
    else
        exception_instruction = *(unsigned long *)(scp-
>sc_pc);
```

Where CAUSE_BD is defined in *<mips/cpu.h>*.

The signal handler may fix the cause of the exception and re-execute the instruction, emulate the instruction and then step over it or perform some non-local goto such as a *longjump()* or an *exit()*.

If corrective action is performed in the signal handler and the instruction that caused the exception would then execute without a further exception, the signal handler simply returns and re-executes the instruction (even when the *branch delay* bit is set).

If execution is to continue after stepping over the instruction that caused the exception the program counter must be advanced. If the *branch delay* bit is set the program counter is set to the target of the branch else it is incremented by 4.

This can be done with the following code sequence:

```
if(scp->sc_cause & CAUSE_BD)
    emulate_branch(scp, branch_instruction);
else
    scp->sc_pc += 4;
```

Emulate_branch() modifies the program counter value in the sigcontext structure to the target of the branch instruction. See *emulate_branch(3)* for more details.

For SIGFPE's generated by floating-point instructions (*code* == 0) the *floating-point control and status* register at the time of the exception is also saved in the sigcontext structure (*sc_fpc_csr*). This register has the information on which exceptions have occurred. When a signal handler is entered the register contains the value at the time of the exception but with the *exceptions bits* cleared. On a return from the signal handler the exception bits in the floating-point control and status register are also cleared so that another SIGFPE does not occur (all other bits are restored from *sc_fpc_csr*).

For SIGSEGV and SIGBUS errors the faulting virtual address is saved in *sc_badvaddr* in the signal context structure.

The SIGTRAP's caused by **break** instructions noted in the above table and all other yet to be defined **break** instructions fill the *code* parameter with the first argument to the **break** instruction (bits 25-16 of the instruction).

Return Value

The previous action is returned on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `signal` subroutine fails and action is not taken if one of the following occurs:

[EINVAL]	The *sig* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3), tty(4)

Name

signal – simplified software signal facilities

Syntax

#include <signal.h>

(*signal(sig, func))()
void (*func)();

Description

The `signal` subroutine is a simplified interface to the more general sigvec(2) facility.

A signal is generated by some abnormal event, initiated by a user at a terminal (quit, interrupt, stop), by a program error (bus error, etc.), by request of another program (kill), or when a process is stopped because it wishes to access its control terminal while in the background. For further information, see `tty`(4). Signals are optionally generated when a process resumes after being stopped, when the status of child process changes, or when input is ready at the control terminal. Most signals cause termination of the receiving process if no action is taken; some signals instead cause the process receiving them to be stopped, or are simply discarded if the process has not requested otherwise. Except for the SIGKILL and SIGSTOP signals, the `signal` call allows signals either to be ignored or to cause an interrupt to a specified location. The following is a list of all signals with names as in the include file < signal.h >:

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction
SIGTRAP	5*	Trace trap
SIGIOT	6*	IOT instruction
SIGEMT	7*	EMT instruction
SIGFPE	8*	Floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGURG	16•	Urgent condition present on socket
SIGSTOP	17+	Stop (cannot be caught or ignored)
SIGTSTP	18+	Stop signal generated from keyboard
SIGCONT	19•	Continue after stop
SIGCHLD	20•	Child status has changed
SIGTTIN	21+	Background read attempted from control terminal
SIGTTOU	22+	Background write attempted to control terminal
SIGIO	23•	I/O is possible on a descriptor (see fcntl(2))
SIGXCPU	24	Cpu time limit exceeded (see setrlimit(2))
SIGXFSZ	25	File size limit exceeded (see setrlimit(2))

SIGVTALRM	26	Virtual time alarm (see setitimer(2))
SIGPROF	27	Profiling timer alarm (see setitimer(2))
SIGWINCH	28•	Window size change
SIGSHORT	29	System V record locking
SIGUSR1	30	User defined signal
SIGUSR2	31	User defined signal
SIGCLD		System V name for SIGCHLD
SIGABRT		X/OPEN name for SIGIOT

The starred signals in the list above cause a core image if not caught or ignored.

If *func* is SIG_DFL, the default action for signal *sig* is reinstated; this default is termination (with a core image for starred signals) except for signals marked with • or +. Signals marked with • are discarded if the action is SIG_DFL; signals marked with + cause the process to stop. If *func* is SIG_IGN the signal is subsequently ignored and pending instances of the signal are discarded. Otherwise, when the signal occurs further occurrences of the signal are automatically blocked and *func* is called.

A return from the function unblocks the handled signal and continues the process at the point it was interrupted. Unlike previous signal facilities, the handler *func* remains installed after a signal has been delivered.

If a caught signal occurs during certain system calls, causing the call to terminate prematurely, the call is automatically restarted. In particular this can occur during a *read* or `write`(2) on a slow device (such as a terminal; but not a file) and during a `wait`(2).

The value of `signal` is the previous (or initial) value of *func* for the particular signal.

After a `fork`(2) or `vfork`(2) the child inherits all signals. The `execve`(2) system call resets all caught signals to the default action; ignored signals remain ignored.

Return Value

The previous action is returned on a successful call. Otherwise, –1 is returned and *errno* is set to indicate the error.

Diagnostics

The `signal` subroutine will fail and no action will take place if one of the following occur:

[EINVAL]	The *sig* is not a valid signal number.
[EINVAL]	An attempt is made to ignore or supply a handler for SIGKILL or SIGSTOP.

Notes (VAX-11)

The handler routine can be declared:

```
handler(sig, code, scp)
```

Here *sig* is the signal number, into which the hardware faults and traps are mapped as defined below. Code is a parameter which is either a constant as given below or, for compatibility mode faults, the code provided by the hardware. The *scp* is a pointer to the *struct sigcontext* used by the system to restore the process context from before

the signal. Compatibility mode faults are distinguished from the other SIGILL traps by having PSL_CM set in the psl.

The following defines the mapping of hardware traps to signals and codes. All of these symbols are defined in < signal.h >:

Hardware condition	Signal	Code
Arithmetic traps:		
Integer overflow	SIGFPE	FPE_INTOVF_TRAP
Integer division by zero	SIGFPE	FPE_INTDIV_TRAP
Floating overflow trap	SIGFPE	FPE_FLTOVF_TRAP
Floating/decimal division by zero	SIGFPE	FPE_FLTDIV_TRAP
Floating underflow trap	SIGFPE	FPE_FLTUND_TRAP
Decimal overflow trap	SIGFPE	FPE_DECOVF_TRAP
Subscript-range	SIGFPE	FPE_SUBRNG_TRAP
Floating overflow fault	SIGFPE	FPE_FLTOVF_FAULT
Floating divide by zero fault	SIGFPE	FPE_FLTDIV_FAULT
Floating underflow fault	SIGFPE	FPE_FLTUND_FAULT
Length access control	SIGSEGV	faulting virtual addr
Protection violation	SIGBUS	faulting virtual addr
Reserved instruction	SIGILL	ILL_PRIVIN_FAULT
Customer-reserved instr.	SIGEMT	
Reserved operand	SIGILL	ILL_RESOP_FAULT
Reserved addressing	SIGILL	ILL_RESAD_FAULT
Trace pending	SIGTRAP	
Bpt instruction	SIGTRAP	
Compatibility-mode	SIGILL	hardware supplied code
Chme	SIGSEGV	
Chms	SIGSEGV	
Chmu	SIGSEGV	

Environment

When your program is compiled using the System V environment the handler function does NOT remain installed after the signal has been delivered.

Also, when a signal which is to be caught occurs during a read(), write(), or ioctl() to a slow device (like a terminal, but not a file); or during a pause(); or wait() that does not return immediately, the signal handler function will be executed, and then the interrupted system call may return a -1 to the calling process with errno set to EINTR.

See Also

kill(1), kill(2), ptrace(2), sigblock(2), sigpause(2), sigsetmask(2), sigstack(2), sigvec(2), setjmp(3), tty(4)

sigprocmask(3)

Name

sigprocmask – examine and change blocked signals (POSIX)

Syntax

#include <signal.h>

int sigprocmask(*how, set, oset***)**
int *how*;
sigset_t **set, *oset*;

Description

The `sigprocmask` system call is used to examine and/or change the calling process's signal mask. If the value of the argument *set* is not NULL, it points to a set of signals that will be used to change the currently blocked set.

The value of the argument *how* indicates the manner in which the set is changed as defined by the following values, defined in <signal.h>:

SIG_BLOCK
The resulting signal set is the union of the current set and the signal set pointed to by the argument *set*.

SIG_UNBLOCK
The resulting signal set is the intersection of the current set and the complement of the signal set pointed to by the argument *set*.

SIG_SETMASK
The resulting signal set is the signal set pointed to by the argument *set*.

If the argument *oset* is not NULL, the previous mask is stored in the space pointed to by *oset*. If the value of the argument *set* is NULL, the process's signal mask is unchanged; thus, the `sigprocmask`(3) function can be used to enquire about currently blocked signals.

The signal masks used as arguments to this function are manipulated using the `sigsetops`(3) functions.

As a system restriction, SIGKILL and SIGSTOP cannot be blocked.

Return Value

A 0 return value indicates a successful call. A –1 return value indicates an error and *errno* is set to indicated the reason.

Diagnostics

The `sigprocmask` function fails and the signal mask remains unchanged if the follow occurs:

[EINVAL] The value of the *how* argument is not equal to one of the defined values.

See Also

kill(2), sigsetmask(2), sigvec(2), sigblock(2), sigsetops(3)

sigsetjmp(3)

Name

sigsetjmp, siglongjmp – nonlocal goto

Syntax

#include <setjmp.h>

sigsetjmp(*env, savemask*)
sigjmp_buf *env*;

siglongjmp(*env, val*)
sigjmp_buf *env*;

Description

These routines deal with errors and interrupts encountered in a low-level subroutine of a program.

The `sigsetjmp` subroutine saves its stack environment in *env* for later use by `siglongjmp`. It returns a value of 0. If the value of the *savemask* argument is not zero, the `sigsetjmp` subroutine also saves the process´ current signal mask as part of the calling environment.

The `siglongjmp` subroutine restores the environment saved by the last call of `sigsetjmp` with the supplied *env* buffer. If the *env* argument was initialized by a call to the `sigsetjmp` subroutine with a nonzero *savemask* argument, the `siglongjmp` subroutine restores the saved signal mask. It then returns in such a way that execution continues as if the call of `sigsetjmp` had just returned the value *val* to the subroutine that invoked `sigsetjmp`, which must not itself have returned in the interim. However, `siglongjmp` cannot cause `sigsetjmp` to return the value 0. If `siglongjmp` is invoked with a *val* of 0, `sigsetjmp` returns a value of 1. All accessible data have values as of the time `siglongjmp` was called.

Restrictions

The `sigsetjmp` subroutine does not save the current notion of whether the process is executing on the signal stack. When you invoke the `siglongjmp` subroutine, the signal stack is left in an incorrect state.

See Also

sigstack(2), sigvec(2), signal(3), sigprocmask(3)

Name

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember – manipulate signal sets (POSIX)

Syntax

#include <signal.h>

int sigemptyset(*set*)
sigset_t **set*;

int sigfillset (*set*)
sigset_t **set*;

int sigaddset(*set,sig*)
sigset_t **set*;
int *sig*;

int sigdelset(*set,sig*)
sigset_t **set*;
int *sig*;

int sigismember(*set,sig*)
sigset_t **set*;
int *sig*;

Description

The `sigsetops`(3) functions manipulate signal sets used by the other POSIX signal functions `sigaction`(3,) `sigprocmask`(3,) `sigsuspend`(3).

The `sigemptyset`(3) function initializes the signal set pointed to by the argument *set* so that all signals are excluded.

The `sigfillset`(3) function initializes the signal set pointed to by the argument *set* so that all signals are included.

The `sigaddset`(3) and `sigdelset`(3) functions respectively add and delete the individual signal specified by the value of the argument *sig* from the signal set pointed to by the argument *set*.

The `sigismember`(3) function tests whether the signal specified by the value of the argument *sig* is a member of the set pointed to by the argument *set*.

Return Value

Upon successful completion, the `sigismember`(3) function returns a value of 1 if the specified signal is a member of the set. If it is not a member of the set, a value of 0 is returned.

If the `sigaddset`(3,) `sigdelset`(3,) or `sigismember`(3) functions fail a –1 value is returned and *errno* is set to indicate the reason.

Diagnostics

The `sigsetops`(3) function will fail and the signal mask will remain unchanged if one of the following occur:

[EINVAL] The value of the *sig* argument is not a valid signal number

See Also

sigprocmask(3), sigaction(3), sigsuspend(3), sigpending(2)

Name

sigsuspend – wait for signal (POSIX)

Syntax

sigsuspend(sigmask)
sigset_t *sigmask;

Description

The `sigsuspend` system call is the POSIX equivalent of the `sigpause`(2) system call. The behavior of this call is as described on the `sigpause`(2) reference page except, the signal mask is manipulated using the `sigsetops`(3) functions.

See Also

sigpause(2), sigaction(3), sigvec(2)

sleep(3)

Name

sleep – suspend execution for interval

Syntax

unsigned
sleep(seconds)
unsigned seconds;

Description

The current process is suspended from execution for the number of seconds specified by the argument. The actual suspension time may be up to 1 second less than that requested, because scheduled wakeups occur at fixed 1-second intervals, and an arbitrary amount longer because of other activity in the system.

The routine is implemented by setting an interval timer and pausing until it occurs. The previous state of this timer is saved and restored. If the sleep time exceeds the time to the expiration of the previous timer, the process sleeps only until the signal would have occurred, and the signal is sent 1 second later.

Return Value

The value returned by `sleep` is the unslept amount(the requested time minus the time actually slept). This return value may be non-zero in cases where the caller had an alarm set to go off earlier than the end of the requested time, or where `sleep` was interrupted due to a caught signal(see ENVIRONMENT below).

Environment

POSIX
SYSTEM_FIVE
When your program is compiled in POSIX or System V mode, the `sleep` will be terminated by any caught signal. The `sleep` function will return following execution of the signal's catching routine.

See Also

setitimer(2), sigpause(2)

Name

statfs, – get file system statistics

Syntax

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/mount.h>

statfs(path, buffer)
char *path;
struct fs_data *buffer;
```

Description

The `statfs` library routine returns up-to-date information about a mounted file system. The *path* is the path name of any file within the mounted file system. The *buffer* is a pointer to an `fs_data` structure as defined in `getmnt`(2).

Return Value

Upon successful completion, a value of `1` is returned. If the file system is not mounted, `0` is returned. Otherwise, `-1` is returned and the global variable *errno* is set to indicate the error.

Diagnostics

The `statfs` library routine fails if one or more of the following are true:

[ENOTDIR] A component of the path prefix of *path* is not a directory.

[EINVAL] *path* contains a character with the high-order bit set.

[ENAMETOOLONG] The length of a component of *path* exceeds 255 characters, or the length of *path* exceeds 1023 characters.

[ENOENT] The file referred to by *path* does not exist.

[EACCES] Search permission is denied for a component of the path prefix of *path*.

[ELOOP] Too many symbolic links were encountered in translating *path*.

[EFAULT] *buffer* or *path* points to an invalid address.

[EIO] An I/O error occurred while reading from the file system.

See Also

getmnt(2), getmountent(3)

Name

staux – routines that provide scalar interfaces to auxiliaries

Syntax

```
#include <syms.h>

long st_auxbtadd(bt)
long bt;

long st_auxbtsize(iaux,width)
long iaux;
long width;

long st_auxisymadd (isym)
long isym;

long st_auxrndxadd (rfd,index)
long rfd;
long index;

long st_auxrndxadd (idn)
long idn;

void st_addtq (iaux,tq)
long iaux;
long tq;

long st_tqhigh_aux(iaux)
long iaux;

void st_shifttq (iaux, tq)
int iaux;
int tq;

long st_iaux_copyty (ifd, psym)
long ifd;
pSYMR psym;

void st_changeaux (iaux, aux)
long iaux;
AUXU aux;

void st_changeauxrndx (iaux, rfd, index)
long iaux;
long rfd;
long index;
```

Description

Auxiliary entries are unions with a fixed length of four bytes per entry. Much information is packed within the auxiliaries. Rather than have the compiler front-ends handle each type of auxiliary entry directly, the following set of routines provide a high-level scalar interface to the auxiliaries:

st_auxbtadd — Adds a type information record (TIR) to the auxiliaries. It sets the basic type (bt) to the argument and all other fields to zero. The index to this auxiliary entry is returned.

st_auxbtsize	Sets the bit in the TIR, pointed to by the *iaux* argument. This argument says the basic type is a bit field and adds an auxiliary with its width in bits.
st_auxisymadd	Adds an index into the symbol table (or any other scalar) to the auxiliaries. It sets the value to the argument that will occupy all four bytes. The index to this auxiliary entry is returned.
st_auxrndxadd	Adds a relative index, RNDXR, to the auxiliaries. It sets the rfd and index to their respective arguments. The index to this auxiliary entry is returned.
st_auxrndxadd_idn	Works the same as *st_auxrndxadd* except that RNDXR is referenced by an index into the dense number table.
st_iaux_copyty	Copies the type from the specified file (ifd) for the specified symbol into the auxiliary table for the current file. It returns the index to the new aux.
st_shifttq	Shifts in the specified type qualifier, tq (see sym.h), into the auxiliary entry TIR, which is specified by the 'iaux' index into the current file. The current type qualifiers shift up one tq so that the first tq (tq0) is free for the new entry.
st_addtq	Adds a type qualifier in the highest or most significant non-tqNil type qualifier.
st_tqhigh_iaux	Returns the most significant type qualifier given an index into the files aux table.
st_changeaux	Changes the iauxth aux in the current file's auxiliary table to aux.
st_changeauxrndx	Converts the relative index (RNDXR) auxiliary, which is specified by iaux, to the specified arguments.

See Also

stfd(3)

Name

stcu – routines that provide a compilation unit symbol table interface

Syntax

```
#include <syms.h>

pCHDRR st_cuinit ()

void st_setchdr (pchdr)
pCHDRR      pchdr;

pCHDRR st_currentpchdr()

void st_free()

long st_extadd (iss, value, st, sc, index)
long iss;
long value;
long st;
long sc;
long index;

pEXTR st_pext_iext (iext)
long    iext;

pEXTR st_pext_rndx (rndx)
RNDXR rndx;

long st_iextmax()

long st_extstradd (str)
char *str;

char *st_str_extiss (iss)
long iss;

long st_idn_index_fext (index, fext)
long index;
long fext;

long st_idn_rndx (rndx)
RNDXR rndx;

pRNDXR st_pdn_idn (idn)
long idn;
RNDXR st_rndx_idn (idn)
long idn;

void st_setidn (idndest, idnsrc)
long idndest;
long idnsrc;
```

Description

The **stcu** routines provide an interface to objects that occur once per object, rather than once per file descriptor (for example, external symbols, strings, and dense numbers). The routines provide access to the current *chdr* (compile time hdr), which represents the symbol table in running processes with pointers to symbol table

sections rather than indices and offsets used in the disk file representation.

A new symbol table can be created with *st_cuinit*. This routine creates and initializes a CHDRR (see *cmplrs/stsupport.h)*. The CHDRR is the current chdr and is used in all later calls.

NOTE

A chdr can also be created with the read routines (see `stio`(3)). The *st_cuinit* routine returns a pointer to the new CHDRR record.

st_currentchdr Returns a pointer the current chdr.

st_setchdr Sets the current chdr to the *pchdr* argument and sets the per file structures to reflect a change in symbol tables.

st_free Frees all constituent structures associated with the current chdr.

st_extadd Lets you add to the externals table. It returns the index to the new external for future reference and use. The *ifd* field for the external is filled in by the current file (see `stfd`(3)). For more details on the parameters, see *sym.h*.

st_pext_iext and st_pext_rndx
Returns pointers to the external, given a index referencing them. The latter routine requires a relative index where the *index* field should be the index in external symbols and the *rfd* field should be the constant ST_EXTIFD. **NOTE**: The externals contain the same structure as symbols (see the *SYMR* and *EXTR* definitions).

st_iextmax Returns the current number of entries in the external symbol table.

The *iss* field in external symbols (the index into string space) must point into external string space.

st_extstradd Adds a null-terminated string to the external string space and returns its index.

st_str_extiss Converts that index into a pointer to the external string.

The dense number table provides a convenience to the code optimizer, generator, and assembler. This table lets them reference symbols from different files and externals with unique densely packed numbers.

st_idn_index_fext Returns a new dense number table index, given an index into the symbol table of the current file (or if *fext* is set, the externals table).

st_idn_rndx Returns a new dense number, but expects a RNDXR (see *sym.h* to specify both the file index and the symbol index rather than implying the file index from the current file. The RNDXR contains two fields: an index into the externals table and a file index *rsyms* can point into the symbol table, as well). The file index is ST_EXTIFD (see *stsupport.h)* for externals.

st_rndx_idn Returns a RNDX, given an index into the dense number table.

st_pdn_idn Returns a pointer to the RNDXR index by the idn argument.

See Also

stfe(3), stfd(3)

Name

stfd – routines that provide access to per file descriptor section of the symbol table

Syntax

```
#include <syms.h>

long st_currentifd ()

long st_ifdmax ()

void st_setfd (ifd)
long ifd;

long st_fdadd (filename)
char *filename;

long st_symadd (iss, value, st, sc, freloc, index)
long iss;
long value;
long st;
long sc;
long freloc;
long index;

long st_auxadd (aux)
AUXU aux;

long st_stradd (cp)
char *cp;

long st_lineadd (line)
long line;

long st_pdadd (isym)
long isym;

long st_ifd_pcfd (pcfd1)
pCFDR pcfd1;

pCFDR st_pcfd_ifd (ifd)
long ifd;

pSYMR st_psym_ifd_isym (ifd, isym)
long ifd;
long isym;

pAUXU st_paux_ifd_iaux (ifd, iaux)
long ifd;
long iaux;

pAUXU st_paux_iaux (iaux)
long iaux;

char *st_str_iss (iss)
long iss;
```

```
char *st_str_ifd_iss (ifd, iss)
long ifd;
long iss;

pPDR st_ppd_ifd_isym (ifd, isym)
long ifd;
long isym;

char * st_malloc (ptr,psize,itemsize,baseitems)
char *ptr;
long *size;
long itemsize;
long baseitems;
```

Description

The **stfd** routines provide an interface to objects handled on a per file descriptor (or fd) level. For example: local symbols, auxiliaries, local strings, line numbers, optimization entries, procedure descriptor entries, and the file descriptors. These routines constitute a group because they deal with objects corresponding to fields in the *FDR* structure.

A fd can be activated by reading an existing one into memory or by creating a new one. The compilation unit routines *st_readbinary* and *st_readst* read file descriptors and their constituent parts into memory from a symbol table on disk.

The *st_fdadd* adds a file descriptor to the list of file descriptors. The *lang* field is initialized from a user specified global *st_lang* that should be set to a constant designated for the language in *symconst.h*. The *fMerge* field is initialized from the user specified global st_merge that specifies whether the file is to start with the attribute of being able to be merged with identical files at load time. The *fBigendian* field is initialized by the `gethostsex`(3) routine, which determines the permanent byte ordering for the auxiliary and line number entries for this file.

The *st_fdadd* adds the null string to the new files string table that is accessible by the constant issNull (0. It also adds the filename to the string table and sets the *rss* field. Finally, the current file is set to the newly added file so that later calls operate on that file.

All routines for fd-level objects handle only the current file unless a file index is specified. The current file can also be set with *st_setfd*.

Programs can find the current file by calling *st_currentifd,* which returns the current index. Programs can find the number of files by calling *st_ifdmax*. The fd routines only require working with indices to do most things. They allow more in-depth manipulation by allowing users to get the compile time file descriptor *(CFDR* see *stsupport.h)* that contains memory pointers to the per file tables (rather than indices or offsets used in disk files). Users can retrieve a pointer to the CFDR by calling *st_pcfd_ifd* with the index to the desired file. The inverse mapping *st_ifd_pcfd* exists, as well.

Each of fd's constituent parts has an add routine: *st_symadd, st_stradd, st_lineadd, st_pdadd,* and *st_auxadd*. The parameters of the add routines correspond to the fields of the added object (see *sym.h*). The *pdadd* routine lets users fill in the isym field only. Further information can be added by directly accessing the procedure descriptor entry.

The add routines return an index that can be used to retrieve a pointer to part of the desired object with one of the following routines: *st_psym_isym, st_str_iss,* and *st_paux_iaux.*

NOTE

These routines only return objects within the current file. The following routines allow for file specification: *st_psym_ifd_isym, st_aux_ifd_iaux,* and *st_str_ifd_iss.*

The *st_ppd_ifd_isym* allows access to procedures through the file index for the file where they occur and the isym field of the entry that points at the local symbol for that procedure.

The return index from st_symadd should be used to get a dense number (see stcu). That number should be the ucode block number for the object the symbol describes.

See Also

stcu(3), stfe(3), sym.h(5), stsupport.h(5)

Name

stfe – routines that provide a high-level interface to basic functions needed to access and add to the symbol table

Syntax

```
#include <syms.h>

long st_filebegin (filename)
char *filename;

long st_endallfiles ()

long st_fileend (idn)
long idn;

long st_blockbegin(iss, value, sc)
long iss;
long value;
long sc;

long st_textblock()

long st_blockend(size)
long size;

long st_procend(idn)
long idn

long st_procbegin (idn)
long idn;

char *st_str_idn (idn)
long idn;

char *st_sym_idn (idn, value, sc, st, index)
long idn;
long *value;
long *sc;
long *st;
long *index;

long st_abs_ifd_index (ifd, index)
long ifd;
long index;

long st_fglobal_idn (idn)
long idn;

pSYMR st_psym_idn_offset (idn, offset)
long idn;
long offset;

long st_pdadd_idn (idn)
long idn;
```

Description

The **stfe** routines provide a high-level interface to the symbol table based on common needs of the compiler front-ends.

st_filebegin — Takes a file name and calls *st_fdadd (see* `stfd`(3)). If it is a new file, a symbol is added to the symbol table that for that file or symbol, and the user supplied routine, *st_feinit,* is called. This allows special file parameters to be initialized. For example, the C front-end adds basic type auxiliaries to each file's aux table so that all variables of that type can refer to a single instance instead of making individual copies of them. The rountine *st_filebegin* returns a dense number that references the symbol added for this file. It tracks files as they appear in a CPP line directive with a stack. It detects (from the order of the CPP directives) that a file ends and calls *st_filend.* If a file is closed with a *st_fileend,* a new instance of the filename is created. For example, multiply included files.

st_fileend — Requires the dense number from the corresponding *st_filebegin call for the file* in question. It then generates an end symbol and patches the references so that the index field of the begin file points to that of one beyond the end file. The end file points to the begin file.

st_endallfiles — Is called at the end of execution to close off all files that have not been ended by previous calls to *st_filebegin.* CPP directives might not reflect the return to the original source file; therefore, this routine can possibly close many files.

st_blockbegin — Supports both language blocks (for example, C's left curly brace blocks), beginning of structures, and unions. If the storage class is scText, it is the former; if it is scInfo, it is one of the latter. The iss (index into string space) specifies the name of the structure/etc, if any.

If the storage class is scText, we must check the result of *st_blockbegin.* It returns a dense number for outer blocks and a zero for nested blocks. The non-zero block number should be used in the BGNB ucode. Users of languages without nested blocks that provide variable declarations can ignore the rest of this paragraph. Nested blocks are two-staged: one stage occurs when the language block is detected and the other stage occurs when the block has content. If the block has content (for example, local variables), the front-end must call *st_textblock* to get a non-zero dense number for the block's BGNB ucode. If the block does not have content and *st_textblock* is not called, the block's *st_blockbegin* and *st_blockend* do not produce block and end symbols.

If it is scInfo, *st_blockbegin* creates a begin block symbol in the symbol table and returns a dense number referencing it. The dense number is necessary to build the auxiliary required to reference the structure/etc. It goes in the aux after the TIR along with a file index. This dense number is also noted in a stack of blocks used by *st_blockend.*

The *st_blockbegin should not be called for* language blocks when the front-end is not producing debugging symbols.

The *st_blockend* requires that blocks occur in a nested fashion. It retrieves the dense number for the most recently started block and creates a corresponding end symbol. As in *fileend,* both the begin and end symbol index fields point at the other end's symbol. If the symbol ends a structure/etc., as determined by the storage class of the begin symbol, the size parameter is assigned to the begin symbol's value field. It is usually the size of the structure or max value of a enum. We only know it at this point. The dense number of the end symbol is returned so that the ucode ENDB can use it. If it is an ignored text block, the dense number is zero and no ENDB should be generated.

In general, defined external procedures or functions appear in the symbols table and the externals table. The external table definition must occur first through the use of a *st_extadd. After that definition, st_procbegin* can be called with a dense number referring to the external symbol for that procedure. It checks to be sure we have a defined procedure (by checking the storage class). It adds a procedure symbol to the symbol table. The external's index should point at its auxiliary data type information (or if debugging is off, indexNil). This index is copied into the regular symbol's index field or a copy of its type is generated (if the external is in a different file than the regular symbol). Next, we put the index to symbol in the external's index field. The external's dense number is used as a block number in ucodes referencing it and is used to add a procedure when in the *st_pdadd_idn.*

st_procend	Creates an end symbol and fixes the indices as in *blockend* and *fileend,* except that the end procedure reference is kept in the begin procedure's aux rather than in the index field (because the begin procedure has a type as well as an end reference). This must be called with the dense number of the procedure's external symbol as an argument and returns the dense number of the end symbol to be used in the END ucode.
st_str_idn	Returns the string associated with symbol or external referenced by the dense number argument. If the symbol was anonymous (for example, there is not a symbol), a (char *), -1 is returned.
st_sym_idn	Returns the same result as *st_str_idn, except that the rest of* by the *idn* are returned in the arguments.
st_fglobal_idn	Returns a 1 if the symbol associated with the specified idn is non-static; otherwise, a 0 is returned.
st_abs_ifd_index	Returns the absolute offset for a dense number. If the symbol is global, the global's index is returned. If the symbol occurred in a file, the sum of all symbols in files occurring before that file and the symbol's index within the file is returned.
st_pdadd_idn	Adds an entry to the procedure table for the *st_proc entry* generated by procbegin. This should be called when the front-end generates code for the procedure in question.

See Also

stcu(3), stfd(3), sym.h(5), stsupport.h(5)

Name

stime – set time

Syntax

int stime (tp)
long *tp;

Description

The `stime` system call sets the system's time and date. The *tp* argument points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Return Value

Upon successful completion, a value of zero (0) is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

Diagnostics

[EPERM] The effective user ID of the calling process is not the superuser.

See Also

gettimeofday(2), time(3)

Name

stio – routines that provide a binary read/write interface to the MIPS symbol table

Syntax

```
#include <syms.h>

long st_readbinary (filename, how)
char *filename;
char how;

long st_readst (fn, how, filebase, pchdr,flags)
long fn;
char how;
long filebase;
pCHDRR pchdr;
long flags;

void st_writebinary (filename, flags)
char *filename;
long flags;

void st_writest (fn, flags)
long fn;
long flags;
```

Description

The CHDRR structure (see **cmplrs/stsupport.h** and the `stcu`(3)). represents a symbol table in memory. A new CHDRR can be created by reading a symbol table in from disk. The *st_readbinary* and *st_readst* routines read a symbol table in from disk.

The routine *st_readbinary* takes the file name of the symbol table and assumes the symbol table header *(HDRR* in **sym.h** occurs at the beginning of the file. The *st_readst* assumes that its file number references a file positioned at the beginning of the symbol table header and that the *filebase* parameter specifies where the object or symbol table file is based (for example, non-zero for archives).

The second parameter to the read routines can be r for read only or a for appending to the symbol table. Existing local symbol, line, procedure, auxiliary, optimization, and local string tables cannot be appended. If they didn't exist on disk, they can be created. This restriction stems from the allocation algorithm for those symbol table sections when read in from disk and follows the standard pattern for building the symbol table.

The symbol table can be read incrementally. If *pchdr is zero, st_readst* assumes that a symbol table has not been read yet; therefore, it reads in the symbol table header and file descriptors. The *flags* argument is a bit mask that defines what other tables should be read. The *t_p** constants for each table, defined in stsupport.h, can be ORed. If *flags* equals -1, all tables are read. If *pchdr* is set, the tables specified by *flags* are added to the tables that have already been read. The *pchdr's value can be taken from st_current_pchdr. See* `stcu`(3.)

Line number entries are encoded on disk; the read routines expand them to longs.

If the version stamp is out of date, a warning message is issued to stderr. If the magic number in the HDRR is incorrect, *st_error* is called. All other errors cause the read routines to read non-zero; otherwise, a zero is returned.

The routines *st_writebinary* and *st_writest* are symmetric to the read routines, excluding the *how* and *pchdr* parameters. The *flags* parameter is a bit mask that defines what table should be written. The *st_p** constants for each table, defined in *stsupport.h,* can be ORed. If *flags* equals -1, all tables are written.

The write routines write sections of the table in the approved order, as specified in the link editor `ld`(1) specification.

Line numbers are compressed on disk.

The write routines start all sections of the symbol table on four-byte boundaries.

If the write routines encounter an error, *st_error* is called. After writing the symbol table, further access to the table by other routines is undefined.

See Also

stcu(3), stfs(3), stfw (3), sym.h(5), sterror(5) stsupport.h(5)

Name

strcoll – string collation comparison

Syntax

int strcoll (*s1*, *s2*)
char **s1*, **s2*;

Description

The `strcoll` function returns an integer less than, equal to, or greater than zero depending on whether the string pointed to by *s1* is lexicographically less than, equal to, or greater than the string pointed to by *s2*.

The `strcoll` function performs the comparison by using the collating information defined in the program's locale, category `LC_COLLATE`.

In the C locale, characters collate as if they are unsigned. In all cases `strcoll` works as if `strxfrm` were called on *s1* and *s2*, and `strcmp` was called on the resulting strings.

International Environment

LC_COLLATE Contains the user requirements for language, territory, and codeset for the character collation format. `LC_COLLATE` affects the behavior of regular expressions and the string collation functions in `strcoll`. If `LC_COLLATE` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strcoll` uses the international language database named in the definition to determine the character collation formatting rules. If `LC_COLLATE` is defined, its definition supercedes the definition of `LANG`.

See Also

string(3), setlocale(3), strxfrm(3), environ(5int)

Name

strftime – convert time and date to string

Syntax

#include <time.h>

int strftime (*s, maxsize, format, tm*)
char ***s**;
size_t *maxsize*;
char ***format**;
struct tm ***tm**;

Description

The `strftime` function places characters in the array pointed to by *s*. No more than *maxsize* characters are placed into the array. The `format` string controls this process. This string consists of zero or more directives and ordinary characters. A directive consists of a `%` character followed by a character that determines the behavior of the directive. All ordinary characters are copied unchanged into the array, including the terminating null character.

Each directive is replaced by the appropriate characters as shown in the following table. The characters are determined by the program's locale category `LC_TIME` and the values contained in the structure pointed to by *tm*.

Directive	Replaced by
%a	Locale's abbreviated weekday name
%A	Locale's full weekday name
%b	Locale's abbreviated month name
%B	Locale's full month name
%c	Locale's date and time representation
%d	Day of month as a decimal number (01–31)
%D	Date (%m/%d/%y)
%h	Locale's abbreviated month name
%H	Hour as a decimal number (00–23)
%I	Hour as a decimal number (01–12)
%j	Day of year (001–366)
%m	Number of month (01–12)
%M	Minute number (00–59)
%n	Newline character
%p	Locale's equivalent to AM or PM
%r	Time in AM/PM notation
%S	Second number (00–59)
%t	Tab character
%T	Time (%H/%M/%S)
%U	Week number (00–53), Sunday as first day of week
%w	Weekday number (0[Sunday]–6)
%W	Week number (00–53), Monday as first day of week
%x	Locale's date representation
%X	Locale's time representation

%y	Year without century (00–99)
%Y	Year with century
%Z	Timezone name, no characters if no timezone
%%	%

If a directive is used that is not contained in the table, the results are undefined.

International Environment

LC_TIME Contains the user's requirements for language, territory, and codeset for the time format. `LC_TIME` affects the behavior of the time functions in `strftime`. If `LC_TIME` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strftime` uses the international language database named in the definition to determine the time formatting rules. If `LC_TIME` is defined, its definition supercedes the definition of `LANG`.

Return Value

If the total number of resulting characters, including the terminal null character, is not more than *maxsize*, the `strftime` function returns the total of resultant characters placed into the array pointed to by *s*, not including the terminating null character. In all other cases zero is returned and the contents of the array are indeterminate.

As the `timezone` name is not contained in the *tm* structure the value returned by *%Z* is determined by the `timezone` function, see `ctime`.

See Also

ctime(3), setlocale(3)

Name

strcasecmp, strncasecmp, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strstr, strtok, index, rindex – string operations

Syntax

```
#include <strings.h>
```

or

```
#include <string.h>

strcasecmp(s1, s2)
char *s1, *s2;

strncasecmp(s1, s2, n)
char *s1, *s2;

char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

int strcmp(s1, s2)
char *s1, *s2;

int strncmp(s1, s2, n)
char *s1, *s2;
int n

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;
int n

size_t strlen(s)
char *s;

char *strchr(s, c)
char *s;
int c;

char *strrchr(s, c)
char *s;
int c;

char *strpbrk(s1, s2)
char *s1, *s2;

size_t strspn(s1, s2)
char *s1, *s2;

size_t strcspn(s1, s2)
char *s1, *s2;
```

char *strtok(*s1, s2***)**
char ***s1, *s2*;**

char *index(*s, c***)**
char ***s, c*;**

char *rindex(*s, c***)**
char ***s, c*;**
char *strstr(*s1, s2***)**
char ***s1, *s2*;**

Description

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null character). The functions `strcat, strncat, strcpy,` and `strncpy` subroutines all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

The `strcat` subroutine appends a copy of string *s2* to the end of string *s1*. The `strncat` subroutine copies at most *n* characters. Both return a pointer to the null-terminated result.

The `strcmp` subroutine compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. The `strncmp` subroutine makes the same comparison but looks at at most *n* characters. The `strcasecmp` and `strncasecmp` subroutines are identical in function, but are case insensitive. The returned lexicographic difference reflects a conversion to lower-case.

The `strcpy` subroutine copies string *s2* to *s1*, stopping after the null character has been copied. The `strncpy` subroutine copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result will not be null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

The `strlen` subroutine returns the number of characters in *s*, not including the terminating null character.

The `strstr` subroutine returns a pointer to the first occurrence of s2 (excluding the terminating null character) in s1, or a NULL pointer if s2 does not occur in s1. If `strlen`(s2) is zero, `strstr` returns s1.

The `strchr` (`strrchr`) function returns a pointer to the first (last) occurrence of character *c* in string *s*, or a NULL pointer is *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

The `strpbrk` subroutine returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a NULL pointer if no character from *s2* exists in *s1*.

The `strspn` (`strcspn`) subroutine returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

The `strtok` subroutine considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a null character into *s1* immediately following the returned token. The function keeps track of its position in the string between separate calls, so that subsequent calls (which must be made with the first argument a NULL pointer) will work through the string *s1* immediately following

that token. In this way, subsequent calls will work through the string *s1* until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL pointer is returned.

The `index` (`rindex`) subroutine returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

NOTE

The <string.h> header file is provided for compatibility with System V; both <string.h> and <strings.h> refer to the same file.
The `strcmp` and `strncmp` subroutines do unsigned character comparisons.

Name

strxfrm – string transformation

Syntax

size_t strxfrm (*to, from, maxsize*)
char ***to**;
char ***from**;
size_t *maxsize*;

Description

The `strxfrm` function transforms the string pointed to by *from* and places the resulting string into the array pointed to by *to*. The transformation is such that two transformed strings can be ordered by the `strcmp` function as appropriate to the program's locale category `LC_COLLATE`.

The length of the resulting string may be much longer than the original. No more than `maxsize` characters are placed into the resulting string including the terminator. If the transformed string does not exceed `maxsize` characters, the number of characters (less the terminator) is returned. Otherwise the number of characters (less the terminator) in the transformed string is returned and the contents of the array are undefined.

International Environment

LC_COLLATE Contains the user requirements for language, territory, and codeset for the character collation format. `LC_COLLATE` affects the behavior of regular expressions and the string collation functions in `strxfrm`. If `LC_COLLATE` is not defined in the current environment, `LANG` provides the necessary default.

LANG If this environment is set and valid, `strxfrm` uses the international language database named in the definition to determine the character collation formatting rules. If `LC_COLLATE` is defined, its definition supercedes the definition of `LANG`.

See Also

string(3), setlocale(3), strcoll(3), environ(5int)

Name

stty, gtty – set and get terminal state

Syntax

#include <sgtty.h>

stty(fd, buf)
int fd;
struct sgttyb *buf;

gtty(fd, buf)
int fd;
struct sgttyb *buf;

Description

This interface has been superseded by `ioctl`(2).

The `stty` subroutine sets the state of the terminal associated with *fd*. The `gtty` subroutine retrieves the state of the terminal associated with *fd*. To set the state of a terminal the call must have write permission.

The `stty` call is actually ''ioctl(fd, TIOCSETP, buf)'', while the `gtty` call is ''ioctl(fd, TIOCGETP, buf)''. See `ioctl`(2) and `tty`(4) for an explanation.

Return Value

If the call is successful 0 is returned, otherwise –1 is returned and the global variable *errno* contains the reason for the failure.

See Also

ioctl(2), tty(4)

Name

swab – swap bytes

Syntax

swab(from, to, nbytes)
char *from, *to;

Description

The `swab` subroutine copies *nbytes* bytes pointed to by *from* to the position pointed to by *to,* exchanging adjacent even and odd bytes. It is useful for carrying binary data between machines. The *nbytes* should be even.

Name

swap_word, swap_half, swap_filehdr, swap_aouthdr, swap_scnhdr, swap_hdr, swap_fd, swap_fi, swap_sym, swap_ext, swap_pd, swap_dn, swap_opt, swap_aux, swap_reloc, swap_ranlib – swap the sex of the specified structure

Syntax

```
#include <sex.h>
#include <filehdr.h>
#include <aouthdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <symconst.h>
#include <cmplrs/stsupport.h>
#include <reloc.h>
#include <ar.h>

long swap_word( word )
long word;

short swap_half( half )
short half;

void swap_filehdr( pfilehdr, destsex )
FILHDR *pfilehdr;
long destsex;

void swap_aouthdr( paouthdr, destsex )
AOUTHDR *paouthdr;
long destsex;

void swap_scnhdr( pscnhdr, destsex )
SCNHDR *pscnhdr;
long destsex;

void swap_hdr( phdr, destsex )
pHDRR phdr;
long destsex;

void swap_fd( pfd, count, destsex )
pFDR pfd;
long count;
long destsex;

void swap_fi( pfi, count, destsex )
pFIT pfi;
long count;
long destsex;

void swap_sym( psym, count, destsex )
pSYMR psym;
long count;
long destsex;
```

void swap_ext(*pext, count, destsex* **)**
pEXTR pext;
long count;
long destsex;

void swap_pd(*ppd, count, destsex* **)**
pPDR ppd;
long count;
long destsex;

void swap_dn(*pdn, count, destsex* **)**
pRNDXR pdn;
long count;
long destsex;

void swap_opt(*popt, count, destsex* **)**
pOPTR popt;
long count;
long destsex;

void swap_aux(*paux, type, destsex* **)**
pAUXU paux;
long type;
long destsex;

void swap_reloc(*preloc, count, destsex* **)**
struct reloc *preloc;
long count;
long destsex;

void swap_ranlib(*pranlib, count, destsex* **)**
struct ranlib *pranlib;
long count;
long destsex;

Description

All `swapsex` routines that swap headers take a pointer to a header structure to change the byte's sex. The *destsex* argument lets the swapsex routines decide whether to swap bitfields before or after swapping the words in which they occur. If *destsex* equals the hostsex of the machine you are running on, the flip happens before the swap; otherwise, the flip happens after the swap. Although not all routines swap structures containing bitfields, the destsex is required.

The `swap_aux` routine takes a pointer to an *aux* entry and a *type*, which is a ST_AUX_* constant in cmplrs/stsupport.h. The constant specifies the type of the aux entry to change the sex of. All other `swapsex` routines are passed a pointer to an array of structures and a *count* of structures to have the byte sex changed. The routines `swap_word` and `swap_half` are macros declared in *sex.h*. Only the include files that describe the structures being swapped have to be included.

See Also

gethostsex(3)

sysconf(3)

Name

sysconf – get configurable system variables (POSIX)

Syntax

#include <unistd.h>

long sysconf(*name***)**
int *name***;**

Description

The `sysconf` function provides a method for the application to determine the current value of a configurable system limit or option.

The *name* argument represents the system variable to be queried. The following table lists the system variables which may be queried and the corresponding value for the *name* argument. The values for the *name* argument are defined in the <unistd.h> header file.

Variable	name Value
ARG_MAX	_SC_ARG_MAX
CHILD_MAX	_SC_CHILD_MAX
CLK_TCK	_SC_CLK_TCK
NGROUPS_MAX	_SC_NGROUPS_MAX
OPEN_MAX	_SC_OPEN_MAX
PASS_MAX	_SC_PASS_MAX
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_VERSION	_SC_VERSION
_XOPEN_VERSION	_SC_XOPEN_VERSION

Return Value

Upon successful completion, the `sysconf` function returns the current variable value on the system.

If *name* is an invalid value, `sysconf` returns –1 and *errno* is set to indicate the reason. If the variable corresponding to *name* is not defined on the system, `sysconf` returns –1 without changing the value of *errno*.

Diagnostics

The `sysconf` function fails if the following occurs:

[EINVAL] The value of the *name* argument is invalid.

Name

syslog, openlog, closelog – control system log

Syntax

#include <syslog.h>

openlog(ident, logstat)
char *ident;

syslog(priority, message, parameters ...)
char *message;

closelog()

Description

The `syslog` subroutine arranges to write the message onto the system log maintained by `syslog`(8). The message is tagged with priority and it looks like a `printf`(3s) string except that **%m** is replaced by the current error message (collected from `errno`). A trailing new line is added if needed. This message is read by `syslog`(8) and output to the system console or files as appropriate. The maximum number of parameters is 5.

If special processing is needed, `openlog` can be called to initialize the log file. Parameters are *ident* which is prepended to every message, and *logstat* which is a bit field indicating special status; current values are:

LOG_PID
log the process id with each message; useful for identifying daemons.

The `openlog` returns zero on success. If it cannot open the file `/dev/log`, it writes on `/dev/console` instead and returns –1.

The `closelog` can be used to close the log file.

Examples

```
syslog(LOG_SALERT, "who: internal error 23");

openlog("serverftp", LOG_PID);
syslog(LOG_INFO, "Connection from host %d", CallingHost);
```

See Also

syslog(8)

system (3)

Name

system – issue a shell command

Syntax

system(*string*)
char ***string**;

Description

If the *string* argument is the NULL pointer (0) the `system` function tests the accessibility of the command interpreter `sh`(1). The function will return zero for failure to find the command interpretter, and positive if successful.

If the *string* argument is non-NULL the `system` routine causes the *string* to be given to `sh`(1) as input as if the string had been typed as a command at a terminal. The current process waits until the shell has completed, then returns the exit status in the form that `wait`(2) returns.

Diagnostics

Exit status 127 indicates the shell couldn't be executed.

See Also

execve(2), wait(2), popen(3)

Name

time, ftime – get date and time

Syntax

#include <time.h>
time_t time((*long *)0*)

time_t time(*tloc*)
**time_t ****tloc*;

#include <sys/timeb.h>

ftime(*tp*)
**struct timeb ****tp*;

Description

The `time` subroutine returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If *tloc* is nonnull, the return value is also stored in the place to which *tloc* points.

The `ftime` entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
struct timeb
{
        time_t   time;
        unsigned short millitm;
        short    timezone;
        short    dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

See Also

date(1), gettimeofday(2), settimeofday(2), ctime(3)

Name

times – get process times

Syntax

#include <sys/times.h>

clock_t
times(*buffer*)
struct tms *_buffer_;

Description

The `times` subroutine returns time-accounting information for the current process and for the terminated child processes of the current process. All times are in 1/HZ seconds, where HZ is equivalent to 60.

The following structure is returned by `times`:

```
struct tms {
        clock_t  tms_utime;     /* user time */
        clock_t  tms_stime;     /* system time */
        clock_t  tms_cutime;    /* user time, children */
        clock_t  tms_cstime;    /* system time, children */
};
```

The children times are the sum of the children's process times and their children's times.

Return Value

If successful, the function `times` returns the elapsed time since 00:00:00 GMT, January 1, 1970 in units of 1/60's of a second. When the function `times` fails, it returns –1

See Also

time(1), getrusage(2), wait3(2), time(3)

Name

tsearch, tfind, tdelete, twalk – manage binary search trees

Syntax

#include <search.h>

void *tsearch (*key, rootp, compar*)
void ***key**;
void ****rootp**;
int (***compar**)( );

void *tfind (*key, rootp, compar*)
void ***key**;
void ****rootp**;
int (***compar**)( );

void *tdelete (*key, rootp, compar*)
void ***key**;
void ****rootp**;
int (***compar**)( );

void twalk (*root, action*)
void * *root*;
void (***action**)( );

Description

The `tsearch` subroutine is a binary tree search routine generalized from Knuth (6.2.2) Algorithm T. It returns a pointer into a tree indicating where a datum may be found. If the datum does not occur, it is added at an appropriate point in the tree. The *key* points to the datum to be sought in the tree. The *rootp* points to a variable that points to the root of the tree. A NULL pointer value for the variable denotes an empty tree; in this case, the variable will be set to point to the datum at the root of the new tree. The *compar* is the name of the comparison function. It is called with two arguments that point to the elements being compared. The function must return an integer less than, equal to, or greater than zero according as the first argument is to be considered less than, equal to, or greater than the second.

Like `tsearch, tfind` will search for a datum in the tree, returning a pointer to it if found. However, if it is not found, `tfind` will return a NULL pointer. The arguments for `tfind` are the same as for `tsearch.`

The `tdelete` subroutine deletes a node from a binary search tree. It is generalized from Knuth (6.2.2) algorithm D. The arguments are the same as for `tsearch.` The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. The `tdelete` subroutine returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found.

The `twalk` subroutine traverses a binary search tree. The *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf }* VISIT; (defined in the <search.h>

header file), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

Notes

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character.

The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Note that the *root* argument to `twalk` is one level of indirection less than the *rootp* arguments to `tsearch` and `tdelete`.

Return Value

A NULL pointer is returned by `tsearch` if there is not enough space available to create a new node.
A NULL pointer is returned by `tsearch, tfind,` and `tdelete` if *rootp* is NULL on entry.
If the datum is found, both `tsearch` and `tfind` return a pointer to it. If not, `tfind` returns NULL, and `tsearch` returns a pointer to the inserted item.

Restrictions

Results are unpredictable if the calling function alters the pointer to the root.

Diagnostics

A NULL pointer is returned by `tsearch` and `tdelete` if *rootp* is NULL on entry.

See Also

bsearch(3), hsearch(3), lsearch(3)

Name

ttyname, isatty, ttyslot – find terminal name

Syntax

char *ttyname(filedes)

isatty(filedes)

ttyslot()

Description

The `ttyname` subroutine returns a pointer to the null-terminated path name of the terminal device associated with file descriptor *filedes* (this is a system file descriptor and has nothing to do with the standard I/O FILE typedef).

The `isatty` subroutine returns 1 if *filedes* is associated with a terminal device, 0 otherwise.

The `ttyslot` subroutine returns the number of the entry in the `ttys`(5) file for the control terminal of the current process.

Restrictions

The return value points to static data whose content is overwritten by each call.

Diagnostics

The `ttyname` subroutine returns a null pointer (0) if *filedes* does not describe a terminal device in directory `/dev`.

The `ttyslot` subroutine returns 0 if `/etc/ttys` is inaccessible or if it cannot determine the control terminal.

Files

/dev/*
/etc/ttys

See Also

ioctl(2), ttys(5)

Name

ulimit – get and set user limits

Syntax

long ulimit (*cmd, newlimit*)
int *cmd*;
long *newlimit*;

Description

This function provides control over process limits. An explanation of the *cmd* values follow.

Value	Explanation
1	Get the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.
2	Set the process's file size limit to the value of *newlimit*. Any process can decrease this limit, but only a process with an effective user ID of superuser can increase the limit. The `ulimit` system call fails and the limit remains unchanged, if a process with an effective user ID other than superuser attempts to increase its file size limit.
3	Get the maximum possible break value. For further information, see `brk`(2).

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, a value of –1 is returned, and *errno* is set to indicate the error.

Diagnostics

[EINVAL]	Bad value for *cmd*.
[EPERM]	The effective user ID of the calling process is not superuser.

See Also

brk(2), write(2)

Name

utime – set file times

Syntax

#include <sys/types.h>
int utime (*path, times*)
char ***path*;**
struct utimbuf ***times*;**

Description

The *path* points to a pathname naming a file. The `utime` function sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use `utime` in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user can use `utime` this way.

The function `utime` causes the time of the last file status change(st_ctime) to be updated with the current time.

The times in the following structure are measured in seconds since 00:00:00 GMT, January 1, 1970.

```
struct  utimbuf  {
        time_t  actime;     /* access time */
        time_t  modtime;    /* modification time */
};
```

Return Value

Upon successful completion, a value of zero (0) is returned. Otherwise, a value of –1 is returned, and *errno* is set to indicate the error.

Diagnostics

The `utime` function fails, if any of the following is true:

[EACCES]	Search permission is denied by a component of the *path* prefix.
[EACCES]	The effective user ID is not super-user, not the owner of the file, *times* is NULL, and write access is denied.
[EFAULT]	The *times* is not NULL and points outside the process's allocated address space.
[EFAULT]	The *path* points outside the process's allocated address space.
[ENOENT]	The named file does not exist or *path* points to an empty string and the environment defined is POSIX or SYSTEM_FIVE.
[ENOTDIR]	A component of the *path* prefix is not a directory.

[EPERM]	The effective user ID is not a super-user, not the owner of the file, and *times* is not NULL.
[EROFS]	The file system containing the file is mounted read-only.
[ETIMEDOUT]	A connect request or remote file operation failed, because the connected party did not respond after a period of time determined by the communications protocol.

See Also

stat(2)

Name

valloc – aligned memory allocator

Syntax

#include <stdlib.h>

void *valloc(*size***)**
size_t *size***;**

Description

The `valloc` subroutine allocates *size* bytes aligned on a page boundary. It is implemented by calling `malloc`(3) with a slightly larger request, saving the true beginning of the block allocated, and returning a properly aligned pointer.

Diagnostics

The `valloc` subroutine returns a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. The `valloc` subroutine will fail and no additional memory will be allocated if one of the following is true:

[ENOMEM] The limit, as set by `setrlimit`(2), is exceeded.

[ENOMEM] The maximum possible size of a data segment (compiled into the system) is exceeded.

[ENOMEM] Insufficient space exists in the swap area to support the expansion.

Name

varargs – variable argument list

Syntax

#include <varargs.h>

function(**va_alist**)
va_dcl
va_list *pvar*;
va_start(*pvar*);
f = **va_arg**(*pvar*, *type*);
va_end(*pvar*);

Description

This set of macros provides a means of writing portable procedures that accept variable argument lists. Routines having variable argument lists, such as `printf`(3s), that do not use varargs are inherently nonportable, since different machines use different argument passing conventions.

va_alist is used in a function header to declare a variable argument list.

va_dcl is a declaration for **va_alist**. Note that there is no semicolon after **va_dcl.**

va_list is a type which can be used for the variable *pvar*, which is used to traverse the list. One such variable must always be declared.

va_start(pvar) is called to initialize *pvar* to the beginning of the list.

va_arg(*pvar*, *type*) will return the next argument in the list pointed to by *pvar*. The *type* is the type the argument is expected to be. Different types can be mixed, but it is up to the routine to know what type of argument is expected, since it cannot be determined at runtime.

va_end(*pvar*) is used to finish up.

Multiple traversals, each bracketed by **va_start** ... **va_end,** are possible.

Examples

```
#include <varargs.h>
execl(va_alist)
va_dcl
{
        va_list ap;
        char *file;
        char *args[100];
        int argno = 0;

        va_start(ap);
        file = va_arg(ap, char *);
        while (args[argno++] = va_arg(ap, char *))
        B;
        va_end(ap);
        return execv(file, args);
}
```

Restrictions

It is up to the calling routine to determine how many arguments there are, since it is not possible to determine this from the stack frame. For example, `execl` passes a 0 to signal the end of the list. The `printf` command can tell how many arguments are supposed to be there by the format.

Name

vlimit – control maximum system resource consumption

Syntax

#include <sys/vlimit.h>

vlimit(*resource, value*)

Description

This facility has been superseded by `getrlimit`(2).

Limits the consumption by the current process and each process it creates to not individually exceed *value* on the specified *resource*. If *value* is specified as –1, then the current limit is returned and the limit is unchanged. The resources which are currently controllable are:

LIM_NORAISE	Pseudo-limit; if set nonzero then the limits may not be raised. Only the super-user may remove the *noraise* restriction.
LIM_CPU	The maximum number of cpu-seconds to be used by each process.
LIM_FSIZE	The largest single file which can be created.
LIM_DATA	The maximum growth of the data+stack region via `sbrk`(2) beyond the end of the program text.
LIM_STACK	The maximum size of the automatically-extended stack region.
LIM_CORE	the size of the largest core dump that will be created.
LIM_MAXRSS	a soft limit for the amount of physical memory (in bytes) to be given to the program. If memory is tight, the system will prefer to take memory from processes which are exceeding their declared LIM_MAXRSS.

Because this information is stored in the per-process information this system call must be executed directly by the shell if it is to affect all future processes created by the shell; *limit* is thus a built-in command to `csh`(1).

The system refuses to extend the data or stack space when the limits would be exceeded in the normal way. A *break* call fails if the data space limit is reached, or the process is killed when the stack limit is reached. Since the stack cannot be extended, there is no way to send a signal.

A file I/O operation which would create a file which is too large will cause a signal SIGXFSZ to be generated, this normally terminates the process, but may be caught. When the cpu time limit is exceeded, a signal SIGXCPU is sent to the offending process; to allow it time to process the signal it is given 5 seconds grace by raising the cpu time limit.

Restrictions

If LIM_NORAISE is set, then no grace should be given when the CPU time limit is exceeded.

See Also

csh(1)

Name

vtimes – get information about resource utilization

Syntax

vtimes(par_vm, ch_vm)
struct vtimes *par_vm, *ch_vm;

Description

This facility has been superseded by `getrusage`(2).

The `vtimes` routine returns accounting information for the current process and for the terminated child processes of the current process. Either *par_vm* or *ch_vm* or both may be 0, in which case only the information for the pointers which are non-zero is returned.

After the call, each buffer contains information as defined by the contents of the include file /usr/include/sys/vtimes.h:

```
struct vtimes {
      int     vm_utime;               /* user time (*HZ) */
      int     vm_stime;           /* system time (*HZ) */
      /* divide next two by utime+stime to get averages */
      unsigned vm_idsrss;         /* integral of d+s rss */
      unsigned vm_ixrss;          /* integral of text rss */
      int     vm_maxrss;          /* maximum rss */
      int     vm_majflt;          /* major page faults */
      int     vm_minflt;          /* minor page faults */
      int     vm_nswap;           /* number of swaps */
      int     vm_inblk;           /* block reads */
      int     vm_oublk;           /* block writes */
};
```

The *vm_utime* and *vm_stime* fields give the user and system time respectively in 60ths of a second (or 50ths if that is the frequency of wall current in your locality.) The *vm_idrss* and *vm_ixrss* measure memory usage. They are computed by integrating the number of memory pages in use each over cpu time. They are reported as though computed discretely, adding the current memory usage (in 512 byte pages) each time the clock ticks. If a process used 5 core pages over 1 cpu-second for its data and stack, then *vm_idsrss* would have the value 5*60, where *vm_utime+vm_stime* would be the 60. The *vm_idsrss* integrates data and stack segment usage, while *vm_ixrss* integrates text segment usage. The *vm_maxrss* reports the maximum instantaneous sum of the text+data+stack core-resident page count.

The *vm_majflt* field gives the number of page faults which resulted in disk activity; the *vm_minflt* field gives the number of page faults incurred in simulation of reference bits; *vm_nswap* is the number of swaps which occurred. The number of file system input/output events are reported in *vm_inblk* and *vm_oublk* These numbers account only for real I/O. Data supplied by the caching mechanism is charged only to the first process to read or write the data.

See Also

wait3(2), time(3)